IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

HP Docket No.: 10960787- 12 **Patent Application**

Inventor(s): C. Venkatraman, et al. **Group Art: 2142**

Serial No.: 09/865,944 **Examiner: HARRELL, Robert B.**

Filed: May 24, 2001

Title: A System for Providing a Web Page for a Device (as amended)

DECLARATION OF KEITH MOORE IN SUPPORT OF ENABLEMENT OF THE CLAIMS OF THE ABOVE IDENTIFIED APPLICATION

Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Sir:

I, Keith Moore, hereby declare that:

1. I am making this declaration affirming that the specification of the above-identified patent application provides enablement for the claims of the same patent application.
2. Since 1987, I have worked in Hewlett-Packard Laboratories (HP Labs) in Palo Alto, California. My current job title is Distinguished Technologist. I have more than 20 years of experience in distributed systems and networked devices during which I have developed an expertise in communication protocols, distributed object systems, component technologies, embedded systems and embedded distributed systems. My work experience is more fully described in Exhibit 1.
3. For the above-identified patent application, I have reviewed the specification of the parent application 08/740,289 from which

this application claims priority as a continuation, the claims 33-163 as filed in the RCE of July 11, 2005, the portion of the non-final Office Action dated July 29, 2005 relating to the rejection of claims 33-163 under 35 U.S.C. 112, first paragraph, and the Interview Summary for application no. 09/862,622, the comments of which I understand also apply to this patent application.

4. I respectfully assert that one of ordinary skill in the art as of October 23, 1996 would find the specification enabling as to how to make or use a web server which enables control functions to manipulate a claimed device.

5. By October 23, 1996, those of ordinary skill in the art knew that electronic instruments such as function generators, oscilloscopes, spectrum analyzers, gas chromatographs had microcontrollers within them that parsed incoming commands sent over a communication protocol such as IEEE 488.2-1987 that allowed remote control of a device's user interface and measurement functions. Standard user interface functions such as TRIGGER, RESET, SELF-TEST, AUTO-CONFIGURE could be sent to any 488.2 compliant device and processed by the control software and converted to a device specific signal (See Exhibit 2 IEEE Standard Codes, Formats, Protocols, and Common Commands ANSI/IEEE Std 488.2 – 1987, pp. 96 to 97 and 185 to 187 for some illustrative examples).

6. By October 23, 1996, those of ordinary skill in the art were well aware that home appliances such as microwaves, VCRs, printers, and washing machines had microcontrollers within them also for translating user commands into device specific actions, such as setting temperature, recording a show at a certain time, or printing a document, under the control of software. "Echelon's [Neuron] chip is a microcontroller similar to the ones used in the control panel of a microwave oven or the remote control for a TV."

"Universal Intelligence. (Apple Computer Inc. Cofounder Mike Markkula Dreams of Using Neuron Chips," Julie Pitta, *Forbes* March 30, 1992, attached hereto as Exhibit 3.) The Neuron chip included programmable memory which allowed it to be customized for the particular appliance which embodied it.

7. Similarly, it was well understood by one of ordinary skill in the art by October 23, 1996 that a button on a home appliance did not have a direct connection to the mechanical parts of the appliance. For example, buttons such as "CANCEL" or "TEST PAGE" on a printer did not have a direct connection to the mechanical parts such as the inkjet nozzle or charge rollers, but rather that the microcontroller received the signal and caused the action in accordance with the controlling software.

8. By October 23, 1996, it was well-established that printers and other imaging devices such as copiers and fax machines included computer systems with software that controlled the printing process. For example, the printers of the 1980s and 1990s were computer peripherals designed to be controlled by software of an associated computer system. These imaging devices had both local user interface capability and remote user interface capability (such as the "CANCEL" function mentioned above). A user could cancel a job from a user interface on the attached PC or by pushing a button on the printer's front panel. The printer's control software parsed the incoming message and converted the incoming message into device specific signals to accomplish the requested function regardless of whether the command came from a local user interface or from the remote user interface offered on the personal computer.

9. Additionally, by October 23, 1996, printers, copiers and fax machines were interoperable with a variety of different makes of computers which used the same protocols to communicate user

commands to these imaging devices. Thus, microprocessors in such imaging devices translated the user commands received in non-device specific protocols (such as the Printer Job Language http://lprng.sourceforge.net/DISTRIB/RESOURCES/DOCS/pjltkref.pdf 1997, Exhibit 4) into device specific actions or signals necessary to produce a response to the user command. For example on page 103 (reproduced here), the command sequence:

```
<ESC>%-12345X@PJL <CR><LF>
@PJL COMMENT Resets variables to <CR><LF>
@PJL COMMENT control panel settings <CR><LF>
@PJL SET RESOLUTION = 600 <CR><LF>
@PJL SET RET = MEDIUM <CR><LF>
@PJL ENTER LANGUAGE = PCL <CR><LF>
<ESC>E . . . PCL job . . . <ESC>E
<ESC>%-12345X@PJL <CR><LF>
@PJL COMMENT Reset to return to <CR><LF>
@PJL COMMENT control panel settings <CR><LF>
@PJL RESET <CR><LF>
<ESC>%-12345X
```

overrides the default resolution before printing a job and then resets the value after the job is printed. (This command sequence was similar to the command sequence used prior to October 23, 1996.) Therefore, by October 23, 1996, those of ordinary skill in the art were well aware of how to make non-device specific software or hardware on a device translate its user commands to the device specific software or hardware in an understandable format for execution.

10. As evidenced above, one of ordinary skill in the art would understand how to implement the claims of this application based on the specification. Claim 50 is reproduced below as a representative example:

> A system for providing a web page for a device that is a printer, comprising:
>
> (a) a printer web server mechanism, including:

a web server that generates a printer web page which enables control functions for the printer, the web server being embedded in the printer;

a network interface embedded in the printer and coupled to the web server;

a monitor embedded in the printer and coupled to the web server, wherein the monitor controls device-specific functions of the printer and monitors a set of information pertaining to the printer; and

a control/monitor path coupled to the monitor;

(b) a communication path coupled to the network interface; and

(c) a web browser coupled to the communication path for rendering the printer web page.

11. One of ordinary skill in the art would understand from drawings such as Figures 1a and 1b and the written description how to implement a printer web server mechanism comprising a web server that generates a printer web page which enables control functions for the printer. Additionally, the description supports that the embedded web server is coupled to communicate those control functions to a monitor which controls device-specific functions of the printer and monitors a set of information pertaining to the printer.
12. For example, page 5 of the specification provides an implementation example consistent with a system using a processor capable of supporting non-device specific functionality through which instructions for device specific functionality are retrieved and re-directed to device specific control functions. "The web server functionality may be implemented with existing circuitry in a device, such as an exiting [sic] processor, memory,

and input/output circuitry that normally perform device-specific functions, thereby avoiding the extra cost and space required for dedicated web server hardware for the device"

13. Additionally, page 11 of the Specification provides another example of enabling description: "The device 10 includes a processor 200, a memory 210, a set of device-specific hardware 300 along with a set of input/output circuitry 220 that enables communication via the communication path 22. The processor 200 performs device-specific functions for the device 10 in combination with the device-specific hardware 300. The processor 200 is also employed to provide web server functionality in the device 10. In one embodiment, the processor 200 stores the web page 18 in the memory 210 which may also be used to store information associated with normal device-specific functions."

14. In this example, the processor 200 has access to the device specific hardware similar to the customized chips available at the time (See Exhibit 3). However, in addition, device 10 includes in its memory as discussed on page 13 of the Specification a web service functionality, which can be implemented in software executed by the processor 200.

15. By October 23, 1996, one of ordinary skill in the art would be able to implement a web server to receive incoming HTTP requests and obey the basic web server protocols to cause actions such as execution of a program or control of an underlying disk storage system. (See Exhibit 5, Chapters 17-21 of "Managing Internet Information Services" by Cricket Liu et al., published by O'Reilly and copyright December 1994 which provided details for such an implementation.) Based on the teachings of the specification, one of ordinary skill would understand how to program the processor to couple the web server functionality to

control the monitor which controls the device specific functions for a specific claimed device.

16. Therefore, in view of the teachings of the specification of this application, one of ordinary skill in the art would know how to make or use a web server which enables control functions for each claimed device.

17. I declare that all statements made herein of my own knowledge are true and that all statements made on information and belief are believed to be true, and further that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment or both under 18U.S.C.§1001, and that such willful false statements may jeopardize the validity of the above identified patent application or any patent issued thereon.

Executed on: **November 29, 2005,** at **Palo Alto, California**

By:__________________________

Keith Moore

EXHIBIT 1

Work Experience of Keith Moore

1998 to Present: Current Position: Distinguished Technologist and Principal Scientist in the Imaging Systems Laboratory at Hewlett-Packard (HP) Laboratories. Previously, Chief Architect of HP's LaserJet family of products and the HP Indigo digital press.

1994-1998: Created the ORBlite distributed object system and co-authored multiple internal standards on bridging between software object system including the OMG COM/CORBA interworking framework, the OMG Java/CORBA mapping, and the OMG C++ mapping.

1990-1994: Developed protocol analyzers for networked instruments and demonstrated that network connections were capable of controlling and managing high-throughput test and measurement systems.

1987-1990: Designed the measurement server architecture and communication protocols for controlling electronic instruments over network connections (802.3).

1985-1987: Designed the communication protocols to connect PC's and HP's instrument controller to electronic instruments such as oscilloscopes, function generators, and signal counters.

Education

MSCS from Stanford University;
Attended Stanford University as an HP Resident Fellow.
BSEE from Tufts University

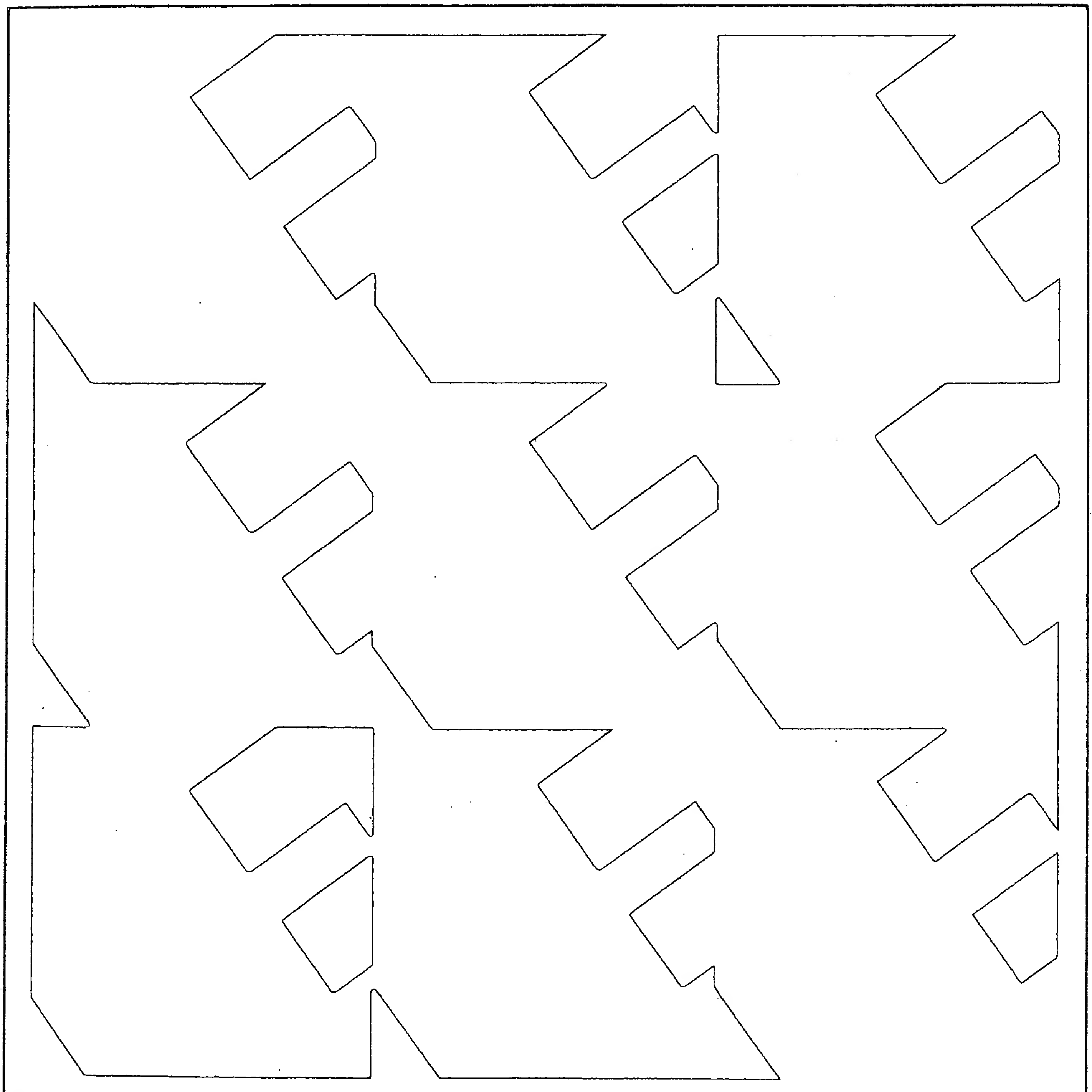
AMERICAN NATIONAL STANDARD

ANSI/IEEE Std 488.2-1987
(Revision and redesignation of
ANSI/IEEE Std 728-1982)

IEEE Standard Codes, Formats, Protocols, and Common Commands
For Use with ANSI/IEEE Std 488.1-1987 IEEE Standard Digital Interface for Programmable Instrumentation

Exhibit 2

ANSI/IEEE Std 488.2-1987



Published by The Institute of Electrical and Electronics Engineers, Inc 345 East 47th Street, New York, NY 10017, USA
April 22, 1988
SH11353

ANSI/IEEE
Std 488.2-1987
(Revision and redesignation of
ANSI/IEEE Std 728-1982)

An American National Standard

IEEE Standard Codes, Formats, Protocols, and Common Commands

For Use with ANSI/IEEE Std 488.1-1987
IEEE Standard Digital Interface for
Programmable Instrumentation

Sponsor

Automated Instrumentation Technical Committee
of the
IEEE Instrumentation and Measurement Society

Approved June 11, 1987

IEEE Standards Board

Approved February 2, 1988

American National Standards Institute

ISBN 471-61871-3

The Institute of Electrical and Electronics Engineers, Inc
345 East 47th Street, New York, NY 10017

10. Common Commands and Queries

This section describes common commands and queries. Descriptive information and compliance requirements are included for each command. Table 10-1 lists alphabetically all the common command headers defined in this standard. Table 10-2 lists the same commands grouped according to function. Syntax of the commands and queries follows the conventions of 7.1.

Table 10-1
IEEE 488.2 Common Command Headers

Mnemonic	Name	Section
*AAD	Accept Address Command	10.1
*CAL?	Calibration Query	10.2
*CLS	Clear Status Command	10.3
*DDT	Define Device Trigger Command	10.4
*DDT?	Define Device Trigger Query	10.5
*DLF	Disable Listener Function Command	10.6
*DMC	Define Macro Command	10.7
*EMC	Enable Macro Command	10.8
*EMC?	Enable Macro Query	10.9
*ESE	Standard Event Status Enable Command	10.10
*ESE?	Standard Event Status Enable Query	10.11
*ESR?	Standard Event Status Register Query	10.12
*GMC?	Get Macro Contents Query	10.13
*IDN?	Identification Query	10.14
*IST?	Individual Status Query	10.15
*LMC?	Learn Macro Query	10.16
*LRN?	Learn Device Setup Query	10.17
*OPC	Operation Complete Command	10.18
*OPC?	Operation Complete Query	10.19
*OPT?	Option Identification Query	10.20
*PCB	Pass Control Back Command	10.21
*PMC	Purge Macro Command	10.22
*PRE	Parallel Poll Register Enable Command	10.23
*PRE?	Parallel Poll Register Enable Query	10.24
*PSC	Power On Status Clear Command	10.25
*PSC?	Power On Status Clear Query	10.26
*PUD	Protected User Data Command	10.27
*PUD?	Protected User Data Query	10.28
*RCL	Recall Command	10.29
*RDT	Resource Description Transfer Command	10.30
*RDT?	Resource Description Transfer Query	10.31
*RST	Reset Command	10.32
*SAV	Save Command	10.33
*SRE	Service Request Enable Command	10.34
*SRE?	Service Request Enable Query	10.35
*STB?	Read Status Byte Query	10.36
*TRG	Trigger Command	10.37
*TST?	Self-Test Query	10.38
*WAI	Wait-to-Continue Command	10.39

Table 10-2
IEEE 488.2 Common Command Groups

Mnemonic	Group	Compliance	Section
*AAD	Auto Configure	Optional†	10.1
*DLF	Auto Configure	Optional†	10.6
*IDN?	System Data	Mandatory	10.14
*OPT?	System Data	Optional	10.20
*PUD	System Data	Optional	10.27
*PUD?	System Data	Optional	10.28
*RDT	System Data	Optional	10.30
*RDT?	System Data	Optional	10.31
*CAL?	Internal Operations	Optional	10.2
*LRN?	Internal Operations	Optional	10.17
*RST	Internal Operations	Mandatory	10.32
*TST?	Internal Operations	Mandatory	10.38
*OPC	Synchronization	Mandatory	10.18
*OPC?	Synchronization	Mandatory	10.19
*WAI	Synchronization	Mandatory	10.39
*DMC	Macro	Optional†	10.7
*EMC	Macro	Optional†	10.8
*EMC?	Macro	Optional†	10.9
*GMC?	Macro	Optional†	10.13
*LMC?	Macro	Optional†	10.16
*PMC	Macro	Optional†	10.22
*IST?	Parallel Poll	Mandatory if PP1	10.15
*PRE	Parallel Poll	Mandatory if PP1	10.23
*PRE?	Parallel Poll	Mandatory if PP1	10.24
*CLS	Status & Event	Mandatory	10.3
*ESE	Status & Event	Mandatory	10.10
*ESE?	Status & Event	Mandatory	10.11
*ESR?	Status & Event	Mandatory	10.12
*PSC	Status & Event	Optional	10.25
*PSC?	Status & Event	Optional	10.26
*SRE	Status & Event	Mandatory	10.34
*SRE?	Status & Event	Mandatory	10.35
*STB?	Status & Event	Mandatory	10.36
*DDT	Trigger	Optional if DT1	10.4
*DDT?	Trigger	Optional if DT1	10.5
*TRG	Trigger	Mandatory if DT1	10.37
*PCB	Controller	Mandatory if other than C0	10.21
*RCL	Stored Settings	Optional†	10.29
*SAV	Stored Settings	Optional†	10.33

† If any commands in either the Auto Configure, Macro, or Stored Settings groups are implemented then all the commands in that group shall be implemented.

Section 6.1.6.1.1 requires that a **device** shall generate a Command Error if an unimplemented common command is received.

```
(7)       .. do something useful with the number ..
(8)     END
```

B4.4 Device Communications While Waiting for a Measurement. The following example illustrates how a response-device can be programmed to take a measurement and then while the **device** is performing it, allow the **device** to send and receive messages. This capability can be useful when a measurement is very time consuming and there is some useful information that the **controller** may wish to read from the **device** while the measurement is in process. The example uses the *OPC common command to generate the Operation Complete message in the Standard Event Status Register when the measurement completes. This example also uses a service request to eliminate the need to poll the Standard Event Status Register with the "*ESR?" common query command.

Device Communications While Waiting for a Measurement

```
STEP
 (1)  RESET 8                              Send IFC, DCL, *RST.
 (2)  SEND 8; "SETUP"
 (3)  SEND 8; "*SRE 32; *ESE 1"            Enable (ESB) Service Request. Allow OPC bit to
                                             set ESB message.
 (4)  SEND 8; "MEAS; *OPC"                 Begin a measurement.
 (5)  WAIT_SRQ
      ...
      ... Perform other tasks involving the device by
      ... sending <PROGRAM MESSAGE> elements or reading
      ... <RESPONSE MESSAGE> elements. Avoid any of the
      ... following: *CLS, *RST, *SRE, *ESE
      ... or any device-specific command
      ... that has a pending operation.
 (6)  READ STATUS BYTE 8; sts
 (7)  IF (RQS message is TRUE in sts)
        THEN
          BEGIN
            SEND 8; "LAST?"
            RECEIVE 8; num
            .. do something useful with the number ..
          END
        ELSE GO TO "Unexpected Service Request Handling"
```

NOTE: The Standard Event Status Register does not have to be read to determine that the Operation Complete message is TRUE because this is the only message that has been enabled to generate a SRQ. A *CLS or a query of the Standard Event Status Register is required to remove the SRQ if the procedure is to be repeated.

B4.5 Synchronization Using an External Control Signal. The synchronization of a response-device with an application program can be accomplished by use of an external-control-signal. This method may be necessary when an asynchronous event must determine when a measurement is taken. This synchronization can be accomplished using any of the three methods indicated in the previous examples. The only changes required are to add device-specific setup commands which direct the **device** to take measurements only when the external-control-signal is received. (See the "TRG: EXT" command for the hypothetical DVM.)

The measurements read are always valid because the **device** does not send data until it has completed each measurement.

B4.6 Example Involving Simultaneous Trigger of Two Response-Devices. The following example shows how two response-devices can be commanded to take a measurement at the same time. The

IEEE 488.1 Group Execute Trigger (GET) command is used to start the measurement. Both **devices** are assumed to be the hypothetical Digital Volt Meter (DVM) defined earlier. The two **devices** have IEEE 488.1 addresses of 8 and 9. The application requires that a pair of voltages be sampled at the same time.

This example uses the *OPC? query command to verify that each **device** has emptied its Input Buffer and has executed all prior commands. Since the **device's** Input Buffer is known to be empty, the application program also knows that the **device** is in a state where it can immediately execute the Group Execute Trigger (GET) command. Since the two **devices** are the same make and model, this example assumes that the two measurements will be taken close together in time.

If the timing needs to be controlled any closer than can be achieved with a Group Execute Trigger, then the application program will have to use means outside of IEEE 488.1 to accomplish the parallel trigger. This triggering could be done using special external trigger hardware in the **devices**.

Simultaneous Trigger of Two Response-Devices

STEP		
(1)	RESET 8	Initialize (DVM-1).
(2)	RESET 9	Initialize (DVM-2).
(3)	SEND 8; "*SRE 0; SETUP "	Turn off Service Requests Set up (DVM-1).
(4)	SEND 9; "*SRE 0; SETUP"	Turn off Service Requests Set up (DVM-2).
(5)	SEND 8; "*DDT #0 MEAS?"	Set DVM-1 to perform a measurement upon receipt of a Group Execute Trigger.
(6)	SEND 9; "*DDT #0 MEAS?"	Set DVM-2 to perform a measurement upon receipt of a Group Execute Trigger.
(7)	SEND 8; *OPC?	
(8)	RECEIVE 8; num	The number returned and stored in variable num is an ASCII "1", discard it. DVM-1's parser is idle.
(9)	SEND 9; "*OPC?"	
(10)	RECEIVE 9; num	The number returned and stored in variable num is an ASCII "1", discard it. DVM-2's parser is idle.

... At this point the Application program directs other
... components in the system to provide the signals
... that are to be measured.

(11)	TRIGGER 8,9;	This command addresses DVM-1 and DVM-2 to listen and then sends the Group Execute Trigger Message.
(12)	RECEIVE 8; num-1	When the measurement from DVM-1 completes, it is sent to the **controller** and placed in variable num-1.
(13)	RECEIVE 9; num-2	When the measurement from DVM-2 completes, it is sent to the **controller** and placed in variable num-2.

...
... do something useful with the read
... measurement data in num-1 and num-2.
...

NOTES: (1) Even though the measurements were read by the **controller** at different times, they were completed at the same time.
(2) The *DDT commands would be unnecessary if the **device** had implemented the "DT1" subset and simply defined Group Execute Trigger to cause the **device** to take a measurement. If the **device** has been built this way the above example might be changed in the following manner:
The application program would have placed the DVMs in a deferred trigger mode as part of the setup. This mode would keep the DVM from taking a measurement until triggered by a GET. The DVM "MEAS?" query commands would not need to be sent to each DVM at all. The GET would be sent to both **devices** causing parallel measurements. The resultant data would then be read from each DVM.

B5. Generalized Synchronization Independent of Queries

Synchronization with **device** operation can also be done with the *OPC command. This command is identical to the *OPC? query command except that the OPC bit is set in the Standard Event Status Register instead of generating the ASCII "1" as an output message. The *OPC command can be used to synchronize either stimulus-devices or response-devices. The *OPC bit of the event status register can be monitored to determine that all pending operations have been completed.

Assume a hypothetical **device** which can perform three overlapped operations: OP1, OP2, and OP3. Each of these commands are Overlapped Commands as defined in 12.2.

Synchronization Independent of Queries

STEP		
(1)	RESET 19	Send IFC, DCL, *RST.
(2)	SEND 19; "SETUP"	
(3)	SEND 19; "*SRE 32; *ESE 1; *CLS"	Enable (ESB) Service Request. Allow OPC bit to set ESB message. CLEAR Standard Event Status Register.
(4)	SEND 19; "OP1; OP2; OP3; *OPC"	Begin a measurement.
(5)	WAIT_SRQ	

```
...
... Perform other tasks involving the device by
... sending <PROGRAM MESSAGE> elements or reading
... <RESPONSE MESSAGE> elements. Avoid any of the
... following: *CLS, *RST, *SRE, *ESE
... or any Overlapped Command.
...
```

```
(6)  READ STATUS BYTE 19; sts
(7)  IF (RQS message is TRUE in sts)
       THEN
         BEGIN
           ... Device has completed all three
           ... commands: OP1, OP2, and OP3.
           ...
           ... Proceed with application.
           ...
         END
       ELSE GO TO "Unexpected Service Request Handling"
```

NOTE: The Standard Event Status Register does not have to be read to determine that the Operation Complete Message is TRUE because this is the only message that has been enabled to generate a SRQ. However, before the program could be written as a loop, the Standard Event Status Register would have to be cleared each time. Clearing can be done by sending a *CLS command or by reading the Standard Event Status Register by using the *ESR? query command.

UNIVERSAL INTELLIGENCE. (APPLE COMPUTER INC. COFOUNDER MIKE MARKKULA DREAMS OF USING NEURON CHIPS) (COMPUTERS/ COMMUNICATIONS)

fb0000020011107do3u000es
By Julie Pitta
806 Words
30 March 1992
Forbes
122
English

Exhibit 3

Apple cofounder Mike Markkula has a big dream: a world of cheap, ubiquitous "neuron" chips.

Along with the better-known Steve Jobs and Steve Wozniak, A.C. (Mike) Markkula founded Apple Computer. Now 50, rich as Croesus and long gone from Apple management, Markkula is innovating again. His Echelon Corp., of Palo Alto, Calif., is working on a smart chip that Markkula says is versatile enough to automate everything from an assembly line to the light fixtures in a home. He calls it, borrowing from the lingo of neural networks, a "neuron" chip.

Echelon's chip is a microcontroller similar to the ones used in the control panel of a microwave oven or the remote control for a TV. What's different about Echelon's chips is that they can talk to each other to create a network of gadgets.

"The number of gadgets with these chips embedded that could be sold in a mature market will be greater than all the semiconductor devices sold today," says Marrkula. "The utopia for Echelon is that you'll be able to walk into any hardware store and any gadget you buy will have one of our chips embedded in the base."

Putting such chips in a coffeemaker, say, or a toaster would enable them to interract. The coffeemaker starts brewing at dawn. Once finished, it signals the toaster to go on.

A fringe idea? Markkula at least speaks with the authority of someone worth $675 million and able to convince several big names to join him. His investors include Motorola and famed venture capitalist Arthur Rock, and his chief executive is ROLM co-founder M. Kenneth Oshman.

"Not every project is as full of risk as this one," says Peter Crisp, and early investors in Apple and now Echelon. "But the market potential is huge." Potential is a word that crops up often at Echelon. The current version of the chip, containing 400,000 transistors, does work, although it's too early to say how well. What's murkier sill is whether manufacturers will want to put these things in their toasters and so on. They may not see the virtue of a world with a universal appliance chip. Today they can get custom-made control chips from a wide variety of suppliers.

None of these doubts assails Markkula. An engineer by training who broke in at Fairchild, he has been toying with the idea of intelligent electronics since his days at Apple. "The more ubiquitous a technology has the potential to become, the more Mike i fascinated by it," says Apple Vice President Albert Eisenstat. Markkula hired the first engineers for his chip six years ago. Early on, Markkula offered a prototype of the chip to Apple and was turned down. "We still think we made the right decision," says Eisenstat.

A year after signing on, a frustrated Oshman came close to folding the project. Markkula toyed with merging Echelon into an established electronics company. Instead, Oshman and Markkula decided in effect to take in partners, persuading Motorola and Toshiba to agree to make and sell the chips, help with research and development and pay Echelon royalties. Motorola also agreed to invest $20 million in Echelon and in return receive a 19% stake.

Echelon has 100 employees, many of them working on software that helps customers program the chips and design intelligent products around them. Like competing control chips, the Echelon neurons contain so-called programmable memory, which allows then to be customized. The Echelon chips, which Motorola is pricing at about $12, have 64 kilobytes of programmable memory and three processors, two of them to send or receive messages.

Potter Electric Signal Co., a small St. Louis manufacturer, has used a batch of Echelon's neurons in smoke alarms, each smart enough to call the fire department..Other manufacturers have designed the neurons into systems to control the environment of a chicken coop, monitor the amount of alcohol poured at a bar or make elevators more intelligent.

Terry Weaver, a vice president at Johnson Controls, the Milwaukee building controls firm, says he is experimenting with Echelon neurons. It's unlikely that he will put an entire office building under the control of one of these things. But Weaver says a cheap chip might nonetheless be useful in a wall switch. If equipped with ancillary electronic sensors that sense the presence of a person in a room, a neuron-equipped switch could cut electric bills.

"If everyone's using the same integrated circuit, the volumes will be enormous--it woul be billions of neurons per year," Oshman says. "The benefit of volume will be that costs

will be low. When they're $1, they will go everywhere." Someday, maybe.

For assistance, access Factiva's Membership Circle.

Printer Job Language Technical Reference Manual

HEWLETT® PACKARD

Edition 10
E1097

HP Part No. 5021-0380
Printed in U.S.A. 10/97

Notice

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or for incidental consequential damages in connection with the furnishing, performance, or use of this material.

Adobe, PostScript, and the PostScript logo are trademarks of Adobe Systems Incorporated, which may be registered in certain jurisdictions. *AppleTalk* is a registered trademark of Apple Computer, Inc. *Centronics* is a U.S. registered trademark of Centronics Data Computer Corporation. *Bi-Tronics* and *HP Explorer Software* are trademarks and *PCL* and *Resolution Enhancement* are registered trademarks of Hewlett-Packard Company. *Epson* is a registered trademark of Epson America, Inc. *Harvard Graphics* is a registered trademark of Software Publishing Corporation. *IBM* is a registered trademark and *ProPrinter* is a trademark of International Business Machines Corporation. *Lotus* and *1-2-3* are registered trademarks of Lotus Development Corporation. *Microsoft* is a registered trademark and *Word, Windows, MS-Mincho,* and *MS-Gothic* are trademarks of Microsoft Corporation. *ESC/P* is a trademark of Seiko-Epson Corporation. *WordPerfect* is a registered trademark of WordPerfect Corporation.

First Edition — September 1992 (P/N 5961-0512)
Second Edition — May 1993 (P/N 5961-0998)
Third Edition — July 1993 (P/N 5961-0603)
Fourth Edition — May 1994 (P/N 5961-0636)
Fifth Edition — September 1994 (P/N 5961-0704)
Sixth Edition — May 1995 (P/N 5010-3996)
Seventh Edition — October 1995 (P/N 5010-3999)
Eighth Edition — May 1996 (P/N 5961-0938)
Ninth Edition — October 1996 (P/N 5021-0328)
Tenth Edition — October 1997 (P/N 5021-0380)

Example: Using the RESET Command

The following example uses a RESET command after the print job to return the features to their previous state:

```
<ESC>%-12345X@PJL <CR><LF0>
@PJL COMMENT Resets variables to <CR><LF>
@PJL COMMENT control panel settings <CR><LF>
@PJL SET RESOLUTION = 600 <CR><LF>
@PJL SET RET = MEDIUM <CR><LF>
@PJL ENTER LANGUAGE = PCL <CR><LF>
<ESC>E . . . PCL job . . . <ESC>E
<ESC>%-12345X@PJL <CR><LF>
@PJL COMMENT Reset to return to <CR><LF>
@PJL COMMENT control panel settings <CR><LF>
@PJL RESET <CR><LF>
<ESC>%-12345X
```

Related Commands:

DEFAULT, INITIALIZE, SET

Managing Internet Information Services

Cricket Liu, Jerry Peek, Russ Jones,
Bryan Buus, and Adrian Nye

with Greg George, Neophytos Iacovou, Jeff LaCoursiere,
Paul Lindner, and Craig Strickland

O'Reilly & Associates, Inc.
103 Morris Street, Suite A
Sebastopol, CA 95472

Managing Internet Information Services

by Cricket Liu, Jerry Peek, Russ Jones, Bryan Buus, & Adrian Nye

Printed in the United States of America.

Editor: Adrian Nye

Production Editor: Ellen Siever

Printing History:

December 1994: First Edition.

This book is printed on acid-free paper with 85% recycled content, 15-20% post-consumer waste. O'Reilly & Associates is committed to using paper with the highest recycled content available consistent with high quality.

ISBN: 1-56592-051-1

In this Chapter:
- *What Is the Web Good For?*
- *Basic Web Concepts*
- *World Wide Web Servers and Browsers*
- *Future Directions*
- *Overview of Web Chapters*

17

Introduction to the World Wide Web

The World Wide Web (known as the Web) is the most graphical Internet service, and it has the most powerful linking abilities. These features have made the Web the fastest growing Internet information service.

The Web allows highlighted words and pictures in a document to link, or point, to other media such as documents, phrases, movie clips, or sound files. The Web can link from any point in a document or image to any point in another document (not just from a menu item to the beginning of a document like Gopher).

With a browser that has a graphical user interface (GUI), the links are followed by simply pointing to the link with the mouse and clicking. These GUI-based versions are probably the easiest of all the Internet services to use and the most fun, because they can mix pictures with text.

The Web was initially developed at CERN, a particle physics laboratory in Geneva, Switzerland. The development work, led by Tim Berners-Lee, started in 1989. Most of the initial work focused on the definition of the HTTP client/server protocol, the development of a sample server, and a programming library called *wwwlib*. In 1992, CERN placed the sample *wwwlib* callable interface in the public domain. That spurred dozens of resourceful organizations and people around the Internet community to start innovating—developing their own Web browsers (programs used to view Web documents), supporting additional platforms, and developing new features. Web servers and Web browsers now support all major computer architectures and operating systems.

The most significant Web software development organization today is the National Center for Supercomputer Applications (NCSA) at the University of Illinois in Urbana-Champaign. Led by Marc Andreessen,* the Software Development Group created a

* Marc Andreessen is now at Mosaic Communications, Inc., a company developing a commercial version of Mosaic.

Web browser called Mosaic. It was built for UNIX, the X Window System, and OSF/Motif. Mosaic was later ported to Microsoft Windows and the Macintosh. Although many others have contributed Web browsers to the Web community, Mosaic remains the only Web browser to uniformly run on the three major desktop environments.

You are already part of the World Wide Web if you currently run an FTP server or a Gopher server. If you have compelling information that the online Web community values, they are probably already accessing your existing information server via the Web.

If you just want to have a personal Web business card (with your picture) and have a URL to include in your *.signature* file for newsgroup posting, you can do that without going to the trouble of setting up a Web server. You can develop your own HTML signature file and put it in your FTP archive. But if you want to set up a rich hypertext environment with extensive linking capability, then you probably should set up a Web server.

The NCSA Web server and related utilities are in the public domain. You can use this Web server with no licensing issues even if you are setting it up specifically as part of a money-making endeavor.

17.1 What Is the Web Good For?

The Web is being used for a wide range of purposes. Here are a few examples:

- Children's science museums are now on the Web, complete with special exhibit information, museum maps, course schedules, ongoing teacher support programs, and more.
- Newspapers are being electronically delivered prior to physical delivery into the community. Users can click to go to restaurant listings, quickly scan restaurants by category, click to look at a restaurant menu, and then click to go see the latest independent review. Readership, however, is global, instead of local. Letters to the editor now come from the other side of the world.
- Business has employed the World Wide Web to deliver product catalogs on demand to potential consumers' desktops. Users scroll through pages online, looking at product photos, clicking to enlarge photos for more detail, checking part numbers and prices—then using the Web to actually select the products and place an order.
- Electronic magazines such as the *Global Network Navigator* (*GNN*) have emerged. These new, innovative magazines are driven by advertising revenue and directly parallel traditional print magazines—they just happen to be electronic and global. But they are developing the capability to accept orders directly from ads, using Web order forms.

- Universities are online. Curious visitors from France can electronically visit a dinosaur exhibit at the Honolulu Community College. History buffs can tour the Vatican exhibit currently being hosted electronically at the University of Illinois.
- International organizations, such as the United Nations, the World Bank, and NATO, are using the Web to deliver information to their constituencies in every country of the world. Many United States agencies are online, as are many of the major proposals submitted to Congress by the Executive Branch—the full National Information Infrastructure (NII) proposal and National Health Care proposal among others. Not just the text of the proposals is available online; relevant photos and diagrams are included. Click on the audio icon, and the recording of the speech that launched the initiative is pulled back to your desktop and played if you have audio capability.
- Local community governments are going online. The City of Palo Alto, California, has a World Wide Web server that ties together the city government, the Chamber of Commerce, the local community historical association, community non-profits, and Palo Alto–based businesses. Many different city maps are online, as is an interactive commuter-train schedule. Users click on their destination and the World Wide Web displays the daily train schedule and the next train to catch.
- Many regions and communities use the World Wide Web as the basis for virtual tourism. Planning a trip to California? The World Wide Web tells you fun things to do and describes local establishments for food and lodging (see Figure 17-1).
- On a personal level, people are creating electronic business cards, with photos and hyperlinks they feel are important. These are the *.signature* files of the Web.

You are probably starting to get the idea that the main limit to what the World Wide Web can do is your imagination.

One downside of the Web is that using its full graphical capability requires the user to have both a GUI-based computer (not a character-based terminal) and an IP Internet connection (not dial-in to an Internet-connected computer). Not everyone has that capability. Also, graphics require a lot of data, so performance suffers over slow Internet links like those of home users.

In a way, the Web's capabilities and flexibility can be a disadvantage. In other services, like Gopher, you format things simply and forget them. The documents served by a Web server can be fine-tuned infinitely and consume more time on presentation (rather than substance) than they're worth. Anyone who has formatted a complicated document on paper probably has experienced this malady.

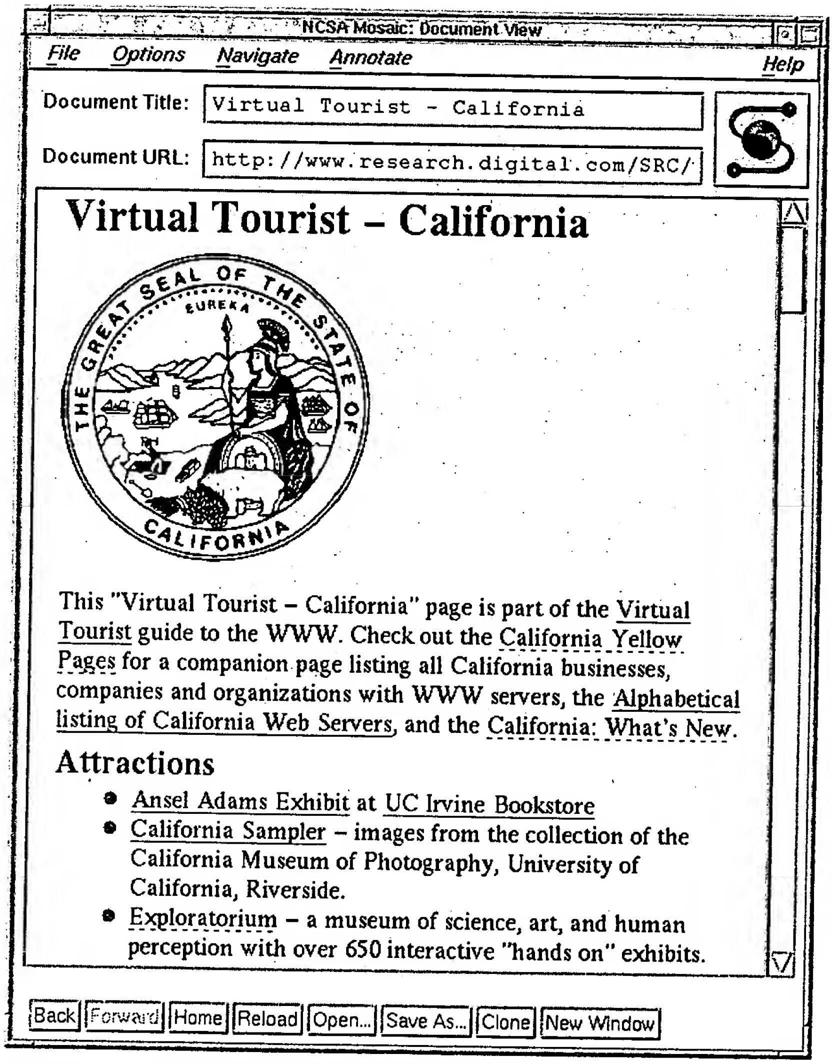


Figure 17-1. California Virtual Tourist document display by Mosiac

17.2 Basic Web Concepts

Here's a brief introduction to the basic concepts of the Web. If you've used the Web much, you might want to just skim this section.

17.2.1 Hyperlinking

To fully appreciate what the World Wide Web does, let's look at an example of hyperlinking in detail. Figure 17-1 showed a screen shot of the Mosaic Web browser displaying the California Virtual Tourist home page. (A home page is the central document on a Web server.) This page includes many embedded hyperlinks that are identified by underlining. The first hyperlink on this page appears as the words Virtual Tourist.

To navigate to the Virtual Tourist document, the user activates the hyperlink with a carriage return or a mouse click. After arriving at the Virtual Tourist document, the user can follow other hyperlinks to more documents or back up to the California Virtual Tourist page. The process of navigating, or following, hyperlinks from document to document is sometimes called surfing.* The power of the Web, and what differentiates it from other environments like Apple's Hypercard, is that hyperlinks can navigate to other documents in the same directory, anywhere on the same Web server, or anywhere on Web servers or other kinds of servers located anywhere in the world.

To illustrate this hyperlinking concept in more detail, Figure 17-2 shows an example of how hyperlinks are used to link documents found on four separate servers.

Server 1 is run by Digital Equipment Corporation, server 2 by State University of New York at Buffalo, server 3 by O'Reilly & Associates, and server 4 by The Exploratorium, a museum in San Francisco. Looking at each of these servers more closely we can put the documents and their hyperlinks into context:

Server 1:

- Document A is the California Virtual Tourist document shown in Figure 17-1. It has embedded hyperlinks from California Yellow Pages to document B on the same server, from Exploratorium to document F on server 4, and from Virtual Tourist to document C on server 2.
- Document B is the California Yellow Pages document. It is linked to from document A.

Server 2:

- Document C is the main Virtual Tourist guide. It is linked to from document A on server 1. It contains a map of the world, with hyperlinks embedded in the world map. It also contains an embedded hyperlink from *Global Network Navigator*'s Traveler's Center to document D on server 3.

Server 3:

- Document D is the *GNN* Traveler's Center. It is linked to from document C on server 2. It has hyperlinks to assorted travel logs and foreign correspondents. It has an embedded hyperlink to document E on the same server.

* Something like "crawling" would make a better metaphor for travelling on a Web. But that wouldn't give the right sense of speed and adventure!

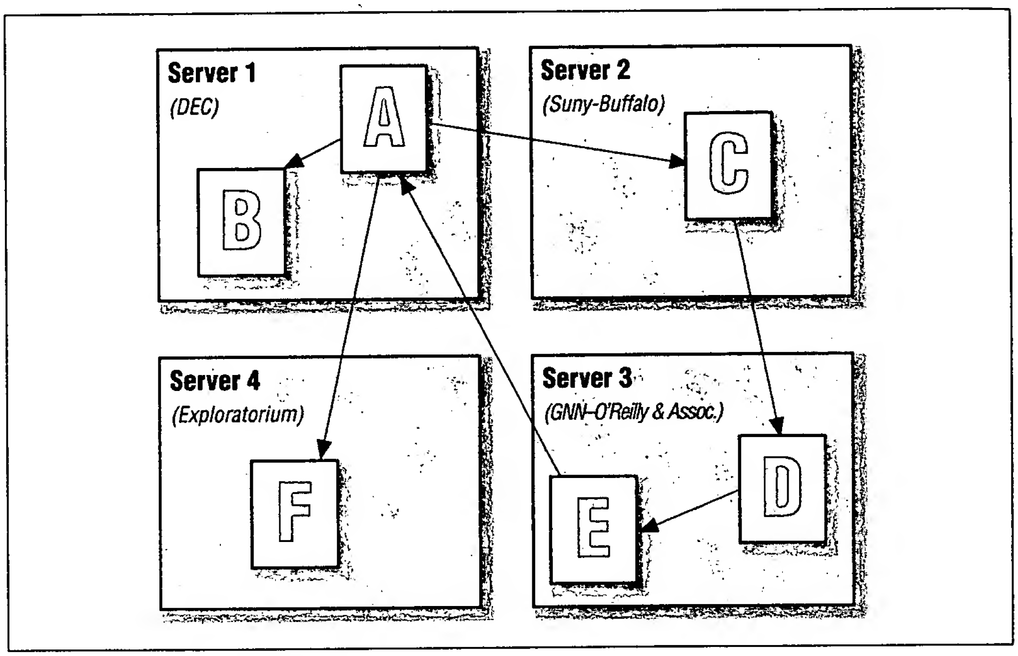


Figure 17-2. A Web of Web servers

- Document E is an index of travel resources by region, country, state, and city. It is linked to from document D. It also contains an embedded hyperlink to document A on server 1, the California Virtual Tourist document.

Server 4:

- Document F is the entry point to the Exploratorium, a hands-on science museum. It is linked to from document A on server 1. It has hyperlinks to local newsletters, an elementary-level science curriculum, and a vast library of photos.

Every document in this example can be further referenced, or pointed to, by other documents in the Web. Each document is a potential starting point for exploring the Web.

With this basic understanding of what the Web does, we can now discuss how it does it.

17.2.2 The HTML Tagging Language

Within the Web, ASCII text documents are marked up with a tagging language called the Hypertext Markup Language (HTML). Such documents are called HTML documents, and usually have a file extension of *.html*. HTML documents can be manually authored with your favorite ASCII text editor, can be converted by filters from other formats, or can be dynamically created at run-time by a Web server or script. The HTML tagging

language is used to describe the structure of the document and hyperlinking information.

HTML describes the structure of a document, but not its exact formatting. Here's a little piece of HTML to give you the flavor (this has been edited and is not a complete HTML document):

```
<H1>NCSA Mosaic Demo Document</H1>
Welcome to NCSA Mosaic, an information browser developed at the National Center
for Supercomputing Applications at the University of Illinois,
Urbana-Champaign.
<P>
This document is an interactive hypermedia tour
of Mosaic's capabilities.
<H2>Introduction</H2>
<UL>
<LI> Mosaic is an Internet-based
<EM>global hypermedia</EM> browser that allows you to discover,
retrieve, and display documents and data from all over the Internet.
<LI> Mosaic is part of the World Wide Web project, a distributed
hypermedia environment originated at CERN and collaborated upon by a
large, informal, and international design and development team.
</UL>
```

We'll describe the complete syntax in Chapter 19, *Authoring for the Web*, but you can easily see headings (<H1> and <H2>), paragraphs (<P>), and a bulleted list (<UL>, <LI> and </UL>). Note that HTML does not specify exactly what size or style of fonts will be used. The browser determines those things. HTML does allow you to identify words to be emphasized (with <EM> in the example above), which might mean italics in a graphical browser but reverse video in a character browser.

HTML also includes tags that ask the user for input, either with simple questions or arbitrarily complex forms.

The current version of HTML is version 2.0. It is based on some of the concepts of SGML, the Standard Generalized Markup Language, which is an ISO standard used for marking up documents for both print and online publication. The next version of the HTML specification, HTML 3.0, will move HTML into full SGML–compliance. HTML 3.0 was previously called HTML+.

Chapter 19, *Authoring for the Web*, is about authoring with HTML.

17.2.3 The URL Concept

Each hyperlink in an HTML document is made up of two components—the anchor text or graphic that, when clicked on, triggers the hyperlink, and the Universal Resource Locator (URL) that describes what to do when the hyperlink is activated. From the user's perspective this means, "when this link is activated, jump to that resource for more information."

The URL describes the protocol used to reach the target server, the host system (or server name) on which the document resides, the directory path to the document, and the document filename. For example, here's an example of a URL to a remote resource. It points to the master list of all public World Wide Web servers throughout the world:

```
http://info.cern.ch/hypertext/DataSources/WWW/Geographical.html
```

This URL means that by using the HTTP (Web) protocol to reach a server called *info.cern.ch* (and using the default port), there is a directory */hypertext/DataSources/WWW* that contains a hypertext document named *Geographical.html*. Every file on the Internet is uniquely addressable by its URL. Besides the HTTP access method, the URL concept also supports other important Internet protocols such as Gopher, FTP, and Telnet. In addition, gateways or enhanced clients can allow the Web to access other types of servers such as Finger and WAIS.

The URLs described above are called absolute URLs, since they completely specify how to get a file. A simplified form of the URL, called a relative URL, is a shorter form for referencing other documents on the same server as the current document. Relative URLs also allow a Web browser to get direct access to files on the system it is running on, without using any server.

This uniform naming scheme for resources is what makes the World Wide Web such a rich information environment, and it's why many people consider World Wide Web browsers the universal Internet access tool.

17.2.4 What Is a World Wide Web Browser?

In the client/server environment of the Web, the control lies with the Web browser. The Web browser's job is to use the initial URL to retrieve a Web document from a Web server, interpret the HTML, and present the document to the user with as much embellishment as the user environment provides. A Web browser written explicitly for a bit-mapped graphic environment can embellish the HTML page to a greater extent than a Web browser written for a character-based environment. A good example of this is the hypertext link. On a graphics workstation, the text used as a hyperlink can be highlighted in a different color or underlined. On a character-based terminal, the same text is probably presented in reverse video.

When the user selects a hypertext link, the process starts all over again—the Web browser uses the URL associated with the hypertext link to request that document, waits for the document to be returned, and then processes and displays the new document.

Figure 17-3 shows the relationship between the Web browser and other Internet information servers discussed in this book.

When interacting with a Gopher server, the Web browser acts like a Gopher client and uses the Gopher protocol. When interacting with an FTP server, the Web browser acts like an FTP client and uses the FTP protocol. When interacting with a Web server, the

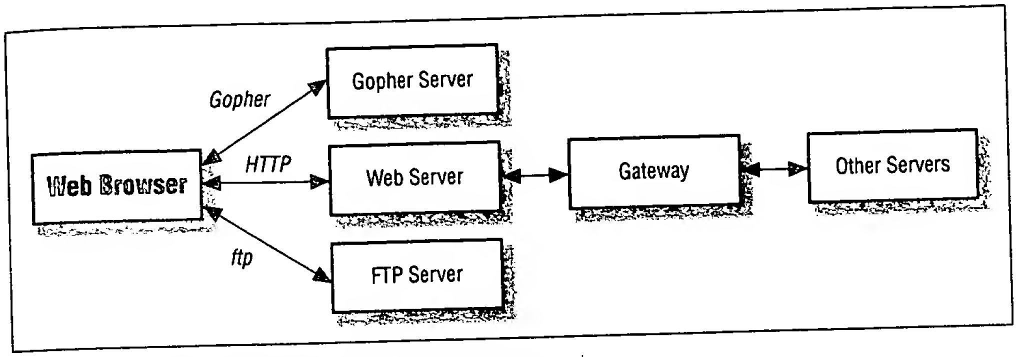


Figure 17-3. The world as seen from a Web browser

Web browser uses the HTTP protocol. Some (but not all) Web browsers can directly access WAIS servers, using the Z39.50 protocol.

Web browsers can also start up *telnet* sessions so that users can access remote Telnet resources and act like news readers to access local news servers.

Many Web browsers also let users do useful things with the current document, such as saving it to disk, sending it as an email message, directing it to the printer, searching through it locally for text strings, or even examining the document's HTML source.

17.2.5 What Is a World Wide Web Server?

The server software used with the World Wide Web is referred to as a Web server. It listens to port 80 (by default). When a client asks for a specific page, the server grabs the page and returns it to the client.

As shown in Figure 17-3, Web servers can execute special scripts that enable them to serve as gateways to other information resources on your system or on the Internet. Many of the most obvious gateway scripts are already available on the Web. You can install them as-is or customize them for your site. You can also develop gateways to access local information systems that might not be directly TCP/IP-accessible.

One kind of script handles input from forms. You can write custom scripts to do anything you want with the data that users enter into the forms.

17.2.6 HTTP

Web browsers and Web servers communicate using the HyperText Transfer Protocol (HTTP). HTTP is a lightweight protocol conceptually similar to the Gopher protocol. Every request for a document from a Web browser to a Web server is a new connection. When a Web browser requests an HTML document from a Web server, the connection is opened, the document is transferred, and the connection is closed.

You will currently see two flavors of HTTP mentioned around the Web, HTTP/0.9 and HTTP/1.0. HTTP/0.9 is being phased out. There is a large degree of compatibility between the two.

HTTP/1.0 supports the negotiation of data types between the Web server and Web browser, by adding MIME (Multimedia Internet Mail Extensions) header information to the protocol. HTML uses the MIME type name of "text" and the MIME subtype name of "html". This is written as:

```
text/html
```

There are many other types supported by Web servers and browsers, such as `image/gif`. With HTTP/1.0, every time a Web browser asks a Web server for a page, it passes along a list of the MIME types it can support. Using this information, the Web server tries to send only MIME types the client supports. The Web server responds first with the MIME type of the data file it is returning, a blank line, and then the actual data file.

The Web browser uses this MIME type information to interpret and display the data file if the type is "text/html" or "text/plain". Otherwise the MIME type information is used to direct the data file to the correct external viewing or playing mechanism. For example, under the X Window System, a Web browser that received a document with a MIME type of image/tiff might start up the program *xv* to display the image.

17.3 World Wide Web Servers and Browsers

Many of the concepts discussed in this chapter are best understood by simply playing with a Web browser to see how it works, how other people have set up their Web servers, and to get ideas about how you might set up your Web server. You will probably also use a browser for accessing documentation about the Web (the document itself is online as part of the World Wide Web), locating and retrieving your Web server kit, and testing the HTML documents you write. Moreover, part of running a Web server is making sure your server works with all of the different Web browsers that your user community uses.

17.3.1 Web Browsers

There are World Wide Web browsers for every conceivable platform, operating system, and graphical user interface (GUI). Web browsers fall into two categories—line-oriented and point-and-click graphical Web browsers. When authoring HTML documents, it helps to have several Web browsers on hand so you can test how your HTML looks with different browsers. Your HTML documents might even look different when displayed on a Macintosh version of Mosaic than it does on a Microsoft Windows version.

We will highlight the major released UNIX Web browsers here and mention some of the other Web browsers that are available or in development. For a complete list of all available browsers, see:

```
http://info.cern.ch/hypertext/WWW/Clients.html
```

17.3.1.1 Lynx

Lynx is a full-screen, character-based Web browser developed at the University of Kansas as a result of efforts to build a campus-wide information system. Figure 19-5 shows what Lynx looks like.

Lynx is a full-featured Web browser that is optimized for cursor-addressable, character-based display devices (such as VT100 terminals, VT100 emulators running on personal computers or Macintoshes, or any other "curses-oriented" display). It uses arrow keys to navigate among embedded HTML links, which are shown in reverse-video. It supports bookmarks to remember your favorite URLs. Lynx also supports forms.

Neat Feature: In interactive mode, the ability to post articles to newsgroups. In non-interactive mode, the ability to filter HTML to formatted ASCII text.

To get Lynx, point your Web browser at:

```
ftp://ftp2.cc.ukans.edu/pub/lynx/
```

This URL points to the FTP archive that contains all of the pre-built binaries for Lynx. There you will find binaries for IBM's RS/6000, DEC Alpha AXP (running OSF/1), SunOS, and DEC (ULTRIX).

Author: Lynx was designed by Lou Montulli, Charles Rezac, and Michael Grobe of Academic Computing Services at the University of Kansas. Lynx was implemented by Lou Montulli, *montulli@mcom.com*. You can send Lynx questions to *lynx-help@ukanaix.cc.ukans.edu* and bug reports to *lynx-bug@ ukanaix.cc.ukans.edu.*

17.3.1.2 MidasWWW

MidasWWW was written for the X Window System and has an OSF/Motif look and feel. It was developed at the Stanford Linear Accelerator Center (SLAC). MidasWWW is similar to Mosaic, with more extensive support for inline graphics. For example, it can display PostScript documents and GIF, JPEG, and TIFF files without using an external viewer. MidasWWW also has some navigation features such as a history tree showing all visited documents.

Neat Feature: MidasWWW blurs the visual presentation between PostScript and HTML.

For more information, see:

```
http://www-midas.slac.stanford.edu/midasv22/introduction.html
```

Author: Tony Johnson, *tony_johnson@slac.stanford.edu*. Questions on MidasWWW should be sent to *MidasWWW@slac.stanford.edu*.

17.3.1.3 Mosaic

Mosaic is the best known of the Web browsers. It was developed at the National Center for Supercomputer Applications at the University of Illinois and is available in pre-built binary form for all major UNIX platforms, plus Apple Macintosh and Microsoft Windows. Figure 17-1 shows what the X Window System version of Mosaic looks like.

On UNIX, Mosaic was written for the X Window System and OSF/Motif. It supports inline graphics in either GIF or XBM format. File types that Mosaic cannot handle inline, such as MPEG movies, sound files, PostScript documents, and JPEG images, are automatically sent to external media players.

Neat Feature: NCSA Mosaic supports forms input as described in Section 20.2.

To get Mosaic, point your Web browser at:

```
ftp://ftp.ncsa.uiuc.edu/Mosaic/
```

This URL brings up an HTML page that takes you to all of the pre-built binaries. There you will find binaries for DEC (OSF/1 V1.3), DEC (MIPS/ULTRIX), HP-UX on Series 700 workstations, IBM's AIX, SGI IRIX, and SunOS. There are two versions for SunOS; which to use depends on how you've configured DNS and NIS.

Authors: NCSA Software Development Group in general, and Eric Bina, *ebina@mcom.com*, and Marc Andreessen, *marca@mcom.com*, in particular. You should direct Mosaic questions to *mosaic-x@ncsa.uiuc.edu*.

17.3.1.4 The Emacs World Wide Web browser

There is a World Wide Web browser mode for Emacs known as the w3-mode extension. This mode is extremely portable—any platform that supports an extensible Emacs supports the w3-mode extension. The w3-mode extension supports multiple fonts, bolding, italics, and the mouse if you are using Lemacs or Epoch with the X Window System. It also has forms support, as specified in the HTML 3.0 specification.

Neat Feature: Compatibility with Mosaic lets you use the same hotlist file, global history file, and personal annotation directory with both Emacs and Mosaic. This could be the best Web browser for people who use Mosaic at work, but still need consistent Web access when they dial in from home over slow lines.

To get the w3-mode extension, point your Web browser at:

```
ftp://moose.cs.indiana.edu/pub/elisp/w3/
```

There you will find source and the documentation for the w3-mode extension.

Author: William Perry, *wmperry@spry.com*.

17.3.1.5 NeXT

The NeXT Web browser was one of the original Web browsers developed at CERN to test concepts in use on the World Wide Web today.

The NeXT Web browser (frozen at V0.17) is available only for the NeXT 68000 workstations. A newer version that will support NeXTstep on other platforms is in development.

Neat Feature: The ability to author HTML and browse from the same Web browser.

To get the NeXT browser, point your Web browser at:

```
ftp://info.cern.ch/pub/www/bin/next/
```

This URL takes you to the FTP archive that contains assorted binaries and documentation for the NeXT Web browser.

Author: Tim Berners-Lee, *Tim.Berners-Lee@cern.ch.*

17.3.1.6 perlWWW

perlWWW is a character-based Web browser written in Perl. It relies on *termcap* for all screen manipulations. You can scroll the screen, select buttons, and press them, all with various keystrokes. It supports two sets of keystroke commands, consistent with Emacs and *vi.*

Neat Feature: It's pretty extensible because it's written in Perl.

To get perlWWW, point your Web browser at:

```
ftp://archive.cis.ohio-state.edu/pub/w3browser/
```

This URL takes you to the FTP archive where the perlWWW browser is kept.

Author: Thomas A. Fine, *fine@cis.ohio-state.edu* of Ohio State University.

17.3.1.7 ViolaWWW

ViolaWWW is an extensible, graphical World Wide Web browser for the X Window System.

Based on the Visual Interactive Object-oriented Language and Applications (Viola) scripting language and toolkit, ViolaWWW provides a way to build relatively complex hypertext applications that are beyond the abilities of the current HTML standard.

Neat Feature: HTML 3.0 support includes container paragraphs, nesting lists, input forms, and tables. Additional extensions include multiple columns, document insertion (client side), and dynamically collapsible/expandable lists.

To get ViolaWWW, point your Web browser at:

```
ftp://ftp.ora.com/pub/www/viola/
```

This URL takes you to the FTP archive containing the SPARCstation binary and the source code for other UNIX platforms.

Author: The initial work was done by the Experimental Computing Facility (XCF) at U.C. Berkeley. Pei Y. Wei, *wei@ora.com* of O'Reilly & Associates is currently driving the development of this Web browser. You can also send email to *viola@ora.com*.

17.3.1.8 Other Web browsers

A number of Web browsers are up and coming. You may want to keep an eye on them.

CERN Line Mode Browser: The Line Mode Browser is the original Web browser developed at CERN. It is a general-purpose information-retrieval tool for people with dumb terminals. In non-interactive mode, the browser is capable of fetching and converting documents and can be used as a filter. In interactive mode, it is similar to Lynx, but doesn't take advantage of the advanced video features found on most terminals. Hyperlinks are identified with a trailing numeric identifier in brackets. For more information, see:

```
http://info.cern.ch/hypertext/WWW/LineMode/Status.html
```

Chimera: Chimera is a Web browser with an X/Athena graphical interface. Chimera can access Web servers, FTP archives, and Gopher servers as well as local files. It also supports forms and inline images. Chimera is popular with users who do not want to use OSF/Motif. For more information, see:

```
ftp://ftp.cs.unlv.edu/pub/chimera/
```

TkWWW Browser/Editor for X11: The entire user interface of this browser is written in *tk*, the X toolkit built on *tcl*. TkWWW is the first browser for the X Window System with the ability to edit HTML. For more information, see:

```
http://info.cern.ch/hypertext/WWW/TkWWW/Status.html
```

In addition to these ten browsers, there are several more PC and Macintosh browsers, not to mention browsers for the more obscure systems. It is a lot to keep up with!

17.3.2 Web Servers

You have three public domain UNIX Web servers to choose from. Each has strengths and weaknesses. You will want to check out the latest versions of each to see which features and enhancements are in the latest revisions. There are also Web servers for other operating systems, but the UNIX Web servers are currently the most flexible and robust. This book describes how to set up the NCSA Web server, because it is the most commonly used.

17.3.2.1 NCSA

The NCSA Web server is a public-domain Web server written in C and designed to be small and fast. There are no licensing restrictions for any use.

The NCSA server is compatible with most HTTP/0.9 and HTTP/1.0 Web browsers. It supports directory aliasing so that documents can be served from any physical directory structure. You can customize your server to execute searches, handle HTML forms, provide clickable image maps, and control user access. The NCSA server also has support for including the output of commands or other files in your HTML documents.

Neat Feature: User-supported directories allow you to let your user community serve HTML documents from their home directories.

To get the NCSA Web server, point your Web browser at:

```
http://hoohoo.ncsa.uiuc.edu/docs/setup/PreCompiled.html
```

This URL takes you to a Web page that points to all of the precompiled *httpd* binaries for all major UNIX platforms.

Author: Rob McCool, *robm@mcom.com*. You can send questions or comments to the current developers of the NCSA Web server at *httpd@ncsa.uiuc.edu*.

17.3.2.2 CERN

The CERN Web server is a public-domain hypertext server written in C. There are no licensing restrictions for any use. The CERN server supports forms, clickable image maps, executable server-side scripts to synthesize documents on the fly, and the ability to plug in index search utilities (via CGI), and it provides access authorization. It also can provide document name-to-filename mapping for longer-lived document URLs.

Neat Feature: The main strength of the CERN server is its ability to provide proxy and caching support. A proxy Web server typically runs on a firewall machine, providing access to the outside world for people inside the firewall. It also can cache documents at the firewall, resulting in faster response times to your internal users.

For more information on the CERN Web server, point your Web browser at:

```
ftp://info.cern.ch/pub/www/bin/
```

This URL takes you to the FTP archive where all of the precompiled *httpd* binaries are stored.

Author: Tim Berners-Lee, *Tim.Berners-Lee@cern.ch* and Ari Luotonen, *luotonen@www.cern.ch*.

17.3.2.3 Plexus

Plexus is a public-domain Web server written in Perl. There are no licensing restrictions for any use. Plexus is designed to be extensible, easy to use, and have good performance. It currently supports both HTTP/0.9 and HTTP/1.0 protocols. To get the Plexus 2.2.1 Web server, point your Web browser at:

```
ftp://austin.bsdi.com/plexus/2.2.1/dist/Plexus-2.2.1.tar.Z
```

Neat Feature: It's written in Perl, so it is easy to modify. However, because Perl is not lightweight, it may not be the best choice for a heavily loaded server.

For the latest information on Plexus, point your Web browser at:

```
http://www.bsdi.com/server/doc/plexus.html
```

Author: Plexus is based on the Perl server developed at the University of Indiana by Marc VanHeyningen, *mvanheyn@cs.indiana.edu*. Plexus is maintained by Tony Sanders, *sanders@bsdi.com*. Tony is quick to point out that this project is not related to BSDI, his employer.

Just as with Web browsers, industrial-strength Web servers are under development at a number of different start-up companies. That is not to say there is anything wrong with these public-domain Web servers, but as always, there is a market for software that is supported, or more scalable and secure.

17.4 Future Directions

What does the future hold for the World Wide Web? The Web is evolving in three major technical directions: HTML enhancements, secure transactions, and uniform resource naming. The Web is also becoming more commercial.

You probably won't understand the significance of some of this without experience running a Web server (or authoring documents for one). You can come back to this after you've read some of the later Web chapters. The main point is that the Web is under rapid development. These future directions also hint at what the Web can't do very well today.

17.4.1 HTML Enhancements

On paper, HTML 2.0 has evolved into HTML 3.0, which is SGML-compliant. HTML 3.0 is described with an SGML Document Type Definition (DTD).

HTML 3.0 provides a rich environment for describing forms, tables, style sheets, and other tagging constructs missing in HTML 2.0. Support for mathematical equations will not make it into the specification until HTML 3.1.

The move towards HTML 3.0 will be extremely important for the part of the Web community that authors documents. Some of the major changes and additions found in HTML 3.0 includes:

New tags
: Allows authors to mark up text as *Footnote*, *Margin*, *Abstract*, *Note*, and *Role*.

Inline graphics
: Allows authors to identify figures (with optional captions above or below the figure) and have text wrapped around either side.

Tables
: Allows authors to identify multi-column, multi-row tables with the ability to mix inline graphics and text in the same table.

Forms
: Beyond what is described in this book, authors will be able to include multiple submit buttons per form.

Document toolbars
: Allows authors to create graphical navigation bars for long documents.

Validation
: Allows validation of HTML using SGML tools, so that you know it's correct without trying it in multiple browsers.

Clickable image maps have been redone to support the image map as part of the HTML document. This deliberately shifts the processing of the image map from the Web server out to the Web browser. It makes image-map processing more responsive to the user, as there is no round trip back to a Web server just to see if the user clicked on the right spot.

Graphical Web browsers will be able to optionally identify the hotspots to the user (a la HyperCard) and change the mouse cursor when moving over active hotspots on the inline graphic. Non-graphical Web browsers will be able to render an ordered list of options, even though they can't show the graphic.

For HTML authors, this means that clickable image maps can include relative URLs and can be easily moved around as part of an HTML document without having to go into elaborate server configuration details.

In layman's terms, HTML 3.0 provides the tagging features needed to fully implement more advanced real-world documents. However, in practice, until more than one Web browser implements the full HTML 3.0 processing definition, HTML 3.0 will not have the critical mass necessary for people to start authoring documents. For the latest information on HTML enhancements, point your Web browser at:

```
http://info.cern.ch/hypertext/WWW/MarkUp/MarkUp.html
```

17.4.2 Secure Transactions

The second major trend is towards providing higher levels of security at the transaction level. Although the current Web technology is adequate to deliver product catalogs to the user's desktop, security mechanisms are not built in to place orders over the Web with credit-card numbers, electronic signatures, and legally binding time stamps. Various efforts have made the first items possible in modified versions of current-generation tools (or with add-on components). But in the long run, these issues are being addressed with the development of a *Secure HTTP* specification.

Secure HTTP will ensure the authenticity of transactions and the confidentiality of information exchanged via HTTP. With a Secure HTTP–enabled application, a user could affix digital signatures that couldn't be repudiated, permitting digital contracts that are legally binding and auditable. In addition, sensitive information such as credit card numbers and bid amounts could be encrypted and securely exchanged. To see this specification, point your Web browser at:

```
http://www.commerce.net/information/standards/drafts/shttp.txt
```

For more general information about Secure HTTP, send an email message to *shttp-info@eit.com.*

Enterprise Integration Technologies (EIT), the National Center for Supercomputing Applications (NCSA) at the University of Illinois, and RSA Data Security have announced agreements to jointly develop and distribute secure versions of NCSA Mosaic and NCSA HTTP Web server based on RSA's public-key cryptography and EIT's Secure HTTP software. The enhancements are being made available to NCSA for public distribution for non-commercial use.

17.4.3 Uniform Naming

The third major trend is in the area of uniform resource naming. As important as the URL concept is to the Web, the URL does have limitations. Foremost is that the URL does not uniquely identify a document—it uniquely identifies an instance of a document. It does not account for mirrored documents and versions of information that are out-of-date.

The naming issue hasn't really been settled yet, but the general direction is to have another identifier called the *Universal Resource Name* (URN). The URN would be a unique document identifier much like the International Standard Book Number (ISBN). Every book published in the world has an ISBN that uniquely identifies its name, author, publisher, publication date, etc. It does not tell you which bookstore or libraries have the book, but it does contain the definitive metadata about the book. A URN would serve the same purpose for electronic documents. The general direction is to have URN servers that, given a URN, would return a list of all known URLs that identify instances of the document. Perhaps they could even return the URL of the instance that is electronically closest to your Web browser over the Internet.

The concept of URN's is not World Wide Web–specific, but does have obvious benefits to the Web. This is all based on the real-world observation that information on the Internet is not completely centralized but, in fact, is replicated and distributed all over the world for an assortment of valid reasons.

As we went to press, we heard that the trend seemed to be shifting toward calling the identifier a URC (Universal Resource Citation). The URC would be used with a URL and would include author, authoritative site, a digital signature or checksum, etc.

17.4.4 Commercialization

Taken as a whole, these technical trends are driving another major shift in the World Wide Web—the march towards commercialization:

- Many of the leading UNIX vendors are starting to bundle Web browsers into their base offerings. Santa Cruz Operation (SCO), Quarterdeck, and Digital Equipment Corporation have all struck licensing agreements with NCSA to bundle Mosaic with their products.
- Other firms, such as SPRY, InfoSeek Corp., Quadralay Corp., Spyglass, and California Software, Inc., are starting to license Web browsers and will be selling them as supported products in the commercial marketplace.
- NETCOM has announced NetCruiser, which is their own Web browser targeted at the home user. NETCOM is currently negotiating with modem manufacturers to bundle NetCruiser software with their products.

All of these marketing thrusts will accelerate the deployment of the Web onto more desktops. This will make the World Wide Web increasingly attractive for information providers and publishers.

The development of secure Web browsers and Web servers will provide a mechanism for electronic commerce. This move towards secure client/server transactions and the ability to exchange money electronically over the Web is laying the foundation for a number of initiatives:

Global Network Navigator (GNN)

Sponsored by O'Reilly & Associates, *GNN* was the first online electronic magazine supported by advertising. Check the *GNN* MarketPlace to get an idea of the evolving commercial environment.

```
http://gnn.com/
```

CommerceNet

Sponsored by Smart Valley, Inc. in the San Francisco Bay Area, *CommerceNet* is intended to jump-start the Silicon Valley electronics industry onto the World Wide Web.

`http://www.commerce.net/`

NetWorth.XET N

Sponsored by Galt Technologies in Pittsburgh, *NetWorth* is intended to provide one-stop shopping for money market funds and financial services.

`http://networth.galt.com/`

MecklerWeb

Sponsored by Mecklermedia in Connecticut, *MecklerWeb* is intended to help the Fortune 1000 exploit the World Wide Web to establish an Internet presence.

`http://www.mecklerweb.com/demo.html`

Each of these has a slightly different focus, but all are intended to quickly ramp up the World Wide Web's use as a business tool. There are dozens of others.

17.5 Overview of Web Chapters

The next few chapters describe how to set up a World Wide Web server, how to author information for the Web, how to create gateways to information services that are not accessible directly from browsers, how to create forms and the scripts that handle form input, and how to control access to your Web server. If you are not a system administrator, you should concentrate on Chapter 19, *Authoring for the Web.*

In this Chapter:
- Setting Up Basic Services
- Web Server Maintenance
- Enabling More Features
- Installation Summary

18

Setting Up a Web Server

Setting up a Web server is not hard if you are familiar with UNIX system management. This chapter describes all the software installation, basic configuration, and maintenance. There are a lot of optional features that you can set up later once you've got the basic server working; they're described in Section 18.3. You should also be sure to read Chapter 21, *Web: Access Control and Security.*

18.1 Setting Up Basic Services

In these sections we describe how to get the NCSA Web server up and running in a basic configuration.

18.1.1 Installing a Precompiled Server

NCSA calls their Web server *httpd* (for HTTP daemon). The current version as of this writing is V1.3.

As of this writing, *httpd* binaries are available for the following platforms:

- IRIS Crimson VGXT running IRIX 4.0.5C
- SPARCserver 690MP running SunOS 4.1.3
- DECstation 5000 running Ultrix 4.2 Rev. 96
- DEC 3000 Model 500 running DEC OSF/1 V1.3
- IBM RS/6000 Model 550 running AIX 3.2.4
- HP 9000 model 730 running HP-UX 9.01

You should be fine with other platforms using the same architecture or an upwardly compatible version of the operating system. The easiest way to get the binaries is to point your Web browser at:

```
http://hoohoo.ncsa.uiuc.edu/docs/setup/PreCompiled.html
```

This URL contains a list of all the currently supported systems. Then you can use your Web browser to retrieve the appropiate kit. The kit contains a pre-built version of *httpd*, default configuration files, pre-built gateway scripts, assorted support scripts, and a collection of icons used for directory indexing.

If you are using the pre-built distribution kit, you can put the kit in */usr/local/src/* and call it *httpd.tar.Z*. Then at the command line you should unpack the distribution with:

```
% cd /usr/local/src
% zcat httpd.tar.Z | tar xvf -
```

When you uncompress and untar the kit using the above command, you will be creating a subdirectory called *httpd_1.3* that contains a *README file*, a pre-built NCSA Web server binary called *httpd*, and several subdirectories:

cgi-bin This directory contains sample gateway scripts and precompiled gateway binaries. If you build your own gateways, this is usually where you will put them. The directory is called *cgi-bin* because gateways use a standard interface with the Web server called the Common Gateway Interface (CGI).

conf This directory contains all the configuration files used to control the operations of your Web server.

icons This directory contains an assortment of icons that are used for directory indexing.

logs This directory is where the access log file and error log file are usually kept.

support This directory contains programs that are used for global and per-directory access control.

If you can't use one of the precompiled packages, you need to grab the NCSA HTTP kit and build it from source. From the Web page listing all of the pre-built binaries, there is a hyperlink that takes you to a page from which you can download the source.

18.1.2 Compilation of NCSA HTTP

If you can use one of the pre-built binary kits mentioned above, you can skip this step completely and go ahead to Section 18.1.3. Otherwise, let's press ahead.

To get the source, point your Web browser at:

```
ftp://ftp.ncsa.uiuc.edu/Web/httpd/Unix/ncsa_httpd/current/httpd_source.tar.Z
```

You can use the following example as a guideline, but keep in mind that the command names and directory paths may be different on your system.

When you retrieve the NCSA HTTP source kit, you should put it in */usr/local/src* and call it *ncsa_source.tar.Z*. Then at the command line:

```
% cd /usr/local/src
% zcat ncsa_source.tar.Z | tar xvf -
```

This step uncompresses and un*tars* the distribution source and puts it in a directory called *httpd_1.3*. The source distribution contains the same directories as the binary distribution described in the previous section, except for the additional *src* directory.

To build NCSA HTTP you must compile two things in separate steps: the *httpd* server and the gateways. First we'll build the *httpd* server. We *cd* to the source directory and make a backup copy of the *Makefile* using the following commands:

```
% cd http_1.3/src
% cp Makefile Makefile-dist
```

Now edit the original *Makefile* to make the following changes. Our general rule is: don't change any *Makefile* variables unless you know you have to. With your text editor:

- Change the CC variable to the name of the compiler you are using on your system.
- Change the CFLAGS variable if you want to use different compiler flags than the default. The default setting is probably OK.
- Add any extra libraries that are needed on your system with the EXTRA_LIBS variable.

If you are compiling for a SunOS 4.1.x environment (which is the default), you can save your changes and exit. Otherwise:

- Comment out the AUX_CFLAGS for SunOS 4.1.x.
- Uncomment the AUX_CFLAGS variable for your system.
- If you don't see your system, then the NCSA Web server probably hasn't been ported to your platform, and you should choose the system type you feel is closest to your actual system.
- Now save your changes and exit.

To build the *httpd* binary and move it into place, use the following commands:

```
% make
```

If this doesn't work, you must edit *src/httpd.h* and then try *make* again. If you have to do this, it is also important that you set the BSD flag correctly to either TRUE or FALSE

according to your operating system. When the *make* works, move the Web server into the top-level directory:

```
% mv httpd ..
```

Let's move on to the CGI scripts. *cd* to the *cgi-src* directory:

```
% cd ../cgi-src
```

You can modify the *Makefile* like you just did the *httpd Makefile*. Then just *make* the scripts:

```
% make
```

This compiles and links all of the CGI scripts and moves them to *../cgi-bin*. Of particular importance is the imagemap CGI script. Keep this step in mind, because you need to change the *imagemap.c* source if you are not using the standard directories.

18.1.3 Site-Specific Configuration

Now that you have the binaries you need, you can configure the NCSA Web server to fit the environment at your site, by editing the three configuration files.* The configuration files are:

httpd.conf	Main server configuration file
srm.con	Server resource configuration file
access.conf	Global access control file (ACF)

The *httpd.conf* and *srm.conf* configuration files are the main ones you'll adjust. You also have to edit the *access.conf* configuration file to match what you specify in *httpd.conf*.

There is also a minor configuration file called *mime.types* that is used to control the mapping of file extensions to MIME data types. It doesn't need to be modified unless you are trying to set up non-standard file-type-to-application correlations. For now, you can ignore the *mime.types* configuration file.

All three of the main configuration files are unloaded into a directory called *conf* and are initially found with a *.conf-dist* file extension. These you can leave alone as a reference and make site-specific copies with just a *.conf* file extension:

```
% cp httpd.conf-dist httpd.conf
% cp srm.conf-dist srm.conf
% cp access.conf-dist access.conf
```

When you modify any of these *.conf* files remember that:

* If you built the *httpd* binary from source, you can controthe HTTP server configuration by editing the *src/httpd.h* file and recompiling. However, it is more convenient to control your NCSA Web server with the configuration files.

- Everything is case-insensitive except pathnames and URLs.
- All comment lines begin with #.
- You can have only one directive per line.
- Extra whitespace is ignored.

The first thing we'll do with these configuration files is show what is needed to quickly get your NCSA Web server up and running for unrestricted public access. We'll also assume at this point that this is your Web server and only you have control of its operation. We will highlight the defaults that matter and describe the absolutely minimal changes needed to bring up your NCSA Web server. Later we'll cover how to customize your configuration.

18.1.3.1 Basic httpd.conf setup

httpd.conf controls how your server actually runs, but not details relating to the files it serves (which are handled in *srm.conf*).

To bring up your NCSA Web server as a standalone daemon for general, unrestricted public access, you should:

- Modify `User` and `Group` to whatever name you want. These define the user id (UID) and group id (GID) that the standalone server uses when it runs. We recommend:

  ```
  User http
  Group WWW
  ```

- Make sure the names exist in */etc/passwd* and */etc/group*, respectively. (The default UID is *nobody* and the default Group is *–1*.)
- Modify `ServerAdmin` to be *webmaster@your.domain* (and make sure you set up that email alias so that mail sent to it gets to you). This email address is given to users so they can report run-time problems with your Web server.
- Change `ServerRoot` to the absolute path to the directory where you plan to put the *httpd* binary. This also determines where other related files will go, so it should not be a directory that already contains files. The default is */usr/local/etc/httpd.*
- Specify `ServerName` to be *www.your.domain*. You will later set up a CNAME alias record to define this hostname.
- Change `ServerType` if you want to run the server under *inetd* instead of standalone. The default is standalone.

We will discuss some of these *httpd.conf* directives more later with the discussion of the feature each controls. For a reference on each *httpd.conf* directive, see Appendix F, *Web: srm.conf Directives.*

18.1.3.2 Basic srm.conf setup

The *srm.conf* configuration file controls where your *httpd* server finds your documents and scripts. To initially bring up your NCSA Web server for unrestricted public access, you should:

- Modify the `DocumentRoot` directive to point to the directory you want to be the top of your document tree. This is the main directory from which *httpd* will serve files. The default is */usr/local/etc/httpd/htdocs.*
- Modify the `UserDir` directive to DISABLED while you set up the server. This prevents users from serving files to the public from their home directories (a potential security risk). The default is *public_html.*
- Modify the `Alias` or `ScriptAlias` directives if you plan to put your *icon* subdirectory or *cgi-bin* subdirectory in a non-default location. The default locations are */usr/local/etc/httpd/icons/* and */usr/local/etc/httpd/cgi-bin/.* Additional `Alias` and `ScriptAlias` directives allow you to make some documents or scripts appear to the user to be in one directory, while you actually keep them in another. These are also known as virtual directories.

Many of the directives found in the *srm.conf* configuration file only make sense in the context of implementing advanced features in your Web server. Those directives will be discussed later with the feature each controls. For a reference on each *srm.conf* directive, see Appendix F, *Web: srm.conf Directives.* See Section 18.2.3 for information about the `Alias` directive.

18.1.3.3 Basic access.conf setup

The global Access Control File (ACF) is called *access.conf.* This configuration controls what type of access Web browsers have to your whole Web server or to certain directories.

The default *access.conf* file is:

```
<Directory /usr/local/etc/httpd/cgi-bin>
    Options Indexes FollowSymLinks
</Directory>

<Directory /usr/local/etc/httpd/htdocs>
     Options Indexes FollowSymLinks
     AllowOverride All
     <Limit GET>
            order allow,deny
            allow from all
     </Limit>
</Directory>
```

These defaults enable all features to all Web browsers without any security precautions. There's a whole chapter (Chapter 21, *Web: Access Control and Security*) about access control, so we won't try to explain all this here. For now, to initially bring up your NCSA Web server for unrestricted public access, you should:

- Modify the first `Directory` directive if your *cgi-bin* directory is in a different location.
- Modify the `Options` directive associated with the *cgi-bin* directory to remove the `Indexes` option. It is not a good idea to let users on the Web randomly look through your *httpd* script directory.
- Modify the second `Directory` directive if necessary so that it matches the path defined with the `DocumentRoot` directive in the *srm.conf* configuration file.
- Modify `AllowOverride` from `All` to `None`. You don't want others on your system or in your HTML authoring community to be overriding your global ACF without regard to security.

18.1.4 Installing the Server

You are now ready to move the *httpd* server and its associated files and directories to the `ServerRoot` prior to starting the Web server. Assuming that you didn't modify the `ServerRootx` location and unpacked the server kit in the same directory that we did, use the following commands to move your *httpd* server into place:

```
% cd /usr/local/src/httpd_1.3
% mkdir /usr/local/etc/httpd
% cp -r httpd conf logs icons cgi-bin /usr/local/etc/httpd
```

Remember, the *logs* directory should be writable by the user your server will run under (as defined by the User directive in the *httpd.conf* configuration file). The easiest way to arrange that is:

```
% cd /usr/local/etc/httpd
% chown http logs
```

18.1.5 Starting the Server

You start your Web server differently depending on whether you want to run it standalone or under *inetd*. Basically, running under *inetd* is a good idea while testing and prototyping, whereas running standalone is better for heavy use.

But before we start it up, here's a summary of the three NCSA *httpd* command-line options. Unlike most other services described in this book, *httpd* has very few options,

because almost everything is controlled in configuration files. When executing *httpd*, three command-line flags are available:

–d directory If you changed ServerRoot in *httpd.conf*, you must also specify the same directory here with *–d*. This controls where *httpd* looks for its configuration files.

–f file This specifies an alternative *httpd.conf* configuration file for *httpd* to read.

–v This just displays the current *httpd* server version. You can execute this while your *httpd* server is running.

18.1.5.1 Standalone startup

If you have used all of the defaults in the configuration files, to start the server, simply execute the binary as *root*:

```
% httpd &
```

httpd should be started from *root* so that it can bind to port 80 (which is a reserved port) and open the three log files. Then it changes UID to the group/user specified by the Group directive and User directive in the *httpd.conf* configuration file, for security reasons.

If you are using a ServerRoot other than */usr/local/etc/httpd*, you need to start the daemon with:

```
% httpd -d /your/server/root &
```

where */your/server/root* is the setting of your ServerRoot directive. You will want to automatically start *httpd* when the system comes up. This is done through modifications to a file in */etc/rc** depending on your system. There's an example in Section 7.3.2.

18.1.5.2 Restarting the standalone httpd

You'll probably change the configuration files, especially while setting up your data and scripts. To reboot the *httpd* server after you have changed any configuration file, use the following command:*

```
% kill -HUP `cat logs/httpd.pid`
```

Alternatively, you could use the *ps* command and *grep* for the process id of the *httpd* server. To make sure you are killing the parent *httpd* process, you should look for the *httpd* with the lowest process id and a parent process id of 1.

* Where *logs/httpd.pid* is the default path and filename specified as PidFile in the *httpd.conf* configuration file. It is relative to the ServerRoot directory.

To test that the *httpd* server restarted successfully, you should check the last line of the *httpd* server's error log (*logs/error_log* by default—see Section 18.2.2). It should say:

```
httpd: successful restart.
```

18.1.6 Startup Under inetd

If you modified your ServerType directive in the *httpd.conf* configuration file, you need to set up *inetd* to run your Web server:

- Edit */etc/services* to add a line which resembles the following:

  ```
  http port-number/tcp
  ```

  `port-number` must match the `Port` directive you specified in the *httpd.conf* configuration file. Normally, it should be 80.

- Edit */etc/inetd.conf* to add a line that resembles the following:

  ```
  http stream tcp nowait nobody /usr/local/etc/httpd/httpd httpd
  ```

  Replace */usr/local/etc/httpd/httpd* with the path to the server binary, and `nobody` with the username you want requests fulfilled as (we suggest *http*). The final reference to *httpd* (only) needs `-d ServerRoot` arguments if you changed `ServerRoot` in the *httpd.conf* configuration file.

- Restart *inetd* by finding the *inetd* process using *ps* and using *kill -HUP* to restart it.

Using this method, you will never need to restart the *httpd* server, as it is restarted when needed by the *inetd* process. Of course, the downside is the performance penalty you will pay on every request to your Web server, because *httpd* has to read all the configuration files at startup.

To take down your service, you need to comment out the line you put in *inetd.conf* and restart *inetd*. You might also want to consider using *xinetd*, an extended version of *inetd* that provides security and access control features. For more information on *xinetd*, see Chapter 28, *xinetd*.

18.1.7 Mapping URLs to Documents

To open an HTML document on your Web server, you need to understand how URLs are mapped against your document tree. The first thing to look at is the minimal URL that reaches your Web server:

```
http://server-name
```

When a URL does not include a directory path or a filename, the Web server either returns the contents of a file called *index.html* in the `ServerRoot` directory or, if the file

is nonexistent, automatically generates a directory index similar to *ls -1* (that's the number one, not lowercase L).*

If you are running your *httpd* server on something other than the default port, you need to include the non-standard port number in the minimal URL. For example, if you run *httpd* on port 8001, the minimal URL to reach your Web server is:

```
http://server-name:8001
```

To retrieve anything other than the root-directory index, you also need to append the virtual name of the document to the URL. The virtual name identifies the path to the document according to the Web server. It is not the absolute pathname of the document on your system. Actually, a virtual name is a virtual directory plus a real filename.

The *httpd* server translates URLs to real directories as follows:

- It looks at the beginning of the URL path for any virtual directories defined by `Alias` or `ScriptAlias` directives in *srm.conf*. If it finds one, it replaces the virtual directory with the real directory and processes the request. Virtual directories let you have separate trees for different kinds of information.
- It looks for a prefix of */~user-name* (slash tilde followed by a valid system user name) in the URL and, if it is found, looks in the specified user's public HTML subdirectory (*public_html*) for the file. If the public HTML subdirectory for the specified user doesn't exist, it returns an error. This step is skipped if `UserDir` is set to DISABLED in *srm.conf*.
- Otherwise, it inserts `DocumentRoot` at the beginning of the path and processes the request.

`Alias`, `ScriptAlias`, and `UserDir` directives are all found in the *srm.conf* configuration file. Each `Alias` or `ScriptAlias` line in *srm.conf* defines a virtual directory. For example, the following default `Alias` line:

```
Alias          /icons/       /usr/local/etc/httpd/icons/
```

defines a virtual directory of */icons/*, which, when it appears in URLs, is mapped by the Web server to */usr/local/etc/httpd/icons/*. For example, the URL:

```
http://server-name/icons/mybitmap.xbm
```

is fulfilled with the file */usr/local/etc/httpd/icons/mybitmap.xbm*.

* *index.html* is the default filename. The actual name can be controlled by the DirectoryIndex directive in the *httpd.conf* configuration file.

18.1.8 Testing Your HTTP Server

The next step is to make sure your Web server is running properly. Use your Web browser to open the minimal URL for your Web server and see what happens. If you have an *index.html* file (one that matches the file identified by the `DirectoryIndex` directive), it should be returned. If not, you should see an index of the filenames at the top of your document tree. If you see either, it means your Web server is up and running.

You can also test your Web server directly by *telnet*ing to the port that *httpd* is listening to and then acting like a Web browser by typing HTTP protocol:

```
% telnet www.ora.com 80
Trying 198.112.208.13 ...
Connected to amber.ora.com.
Escape character is '^]'.
HEAD / HTTP/1.0<CR><CR>

HTTP/1.0 200 OK
Date: Saturday, 20-Aug-94 20:33:28 GMT
Server: NCSA/1.1
MIME-version: 1.0
Content-type: text/html
Last-modified: Tuesday, 02-Aug-94 15:32:19 GMT

Connection closed by foreign host.
```

This shows that the Web server is up and running and its version number. Using *telnet* is also useful when you are trying to debug MIME typing problems.

If you can't get your Web server to respond from your Web browser and a direct *telnet* to your server doesn't produce the results shown above, you should take a look at the access log and the error log to see what they say (see Section 18.2.2).

If you are testing from the same system as your *httpd* server is running on, also try it from another system. If this isn't convenient, you might ask a friend who has access to a different system to test your server for you.

18.1.9 Setting Up Home Pages

Although every document on your Web server is potentially the first thing on your server that someone will see, most often the entry point is the home page. By convention this file is named *home.html*. You can have a single *home.html* document or several (in different directories), depending on whether your Web server serves one purpose or is used by multiple unrelated people or organizations. The next chapter describes how to create HTML documents in general. Here, we'll just mention the conventions for home pages.

When constructing a home page you should:

- Sign your work. This is usually done by placing a hyperlink to your HTML signature page at the bottom of your home-page document. We will do this in our sample HTML document in Chapter 19, *Authoring for the Web*, and usually do it on every HTML document. This allows people to report bugs and updates.
- Date your home page with a creation or modification date. We generally like to do this on every HTML document, not just the home page.
- Give the status of the Web server. Indicate whether it is under construction or is stable. Users will use the server's status to decide whether or not to send you email when they find a broken hyperlink. It is fine to start out with a Web server that is "under construction."

You should place a single home page at the top of your document tree in order to have your home page URL as short as possible. For example, the URL for Digital Equipment Corporation's home page could be:

```
http://www.digital.com/home.html
```

To shorten this even further, so that *home.html* does not need to be specified, you can either copy *home.html* into *index.html*, or set your `DirectoryIndex` to be *home.html*.

If your Web server has multiple uses, you will probably have multiple home pages located in different parts of the document tree. In this case, individual home-page URLs are based on where they are located in the document tree. You may want to manually write the *index.html* file in the root directory as an HTML document that gives a structured description of all the home pages hosted by your system.

18.1.10 Delegating Document Tree Management

So far, we have assumed that you have exclusive control of the document tree and its contents. Let's look at how you can release content maintenance to your HTML authoring community, either by project or by individual.

If your Web server supports different organizations or functions, you may want to give each a subdirectory in the document tree that they can manage themselves. You don't have to modify any of the directives in the configuration files to do this. You can control which users on your local system can write and modify different sections of the document tree by using standard UNIX file and directory permissions.

If any of the organizations want to use image maps, gateways, or access control, they need to coordinate it with you. You can also activate specific server features on an organization-by-organization basis.

If your Web server needs to present documents authored by the general user population on your system, you can let individuals serve HTML documents out of a special directory in their personal home directories. This is controlled by the `UserDir` directive

in the *srm.conf* configuration file. If `UserDir` is set to *public_html*, that tells *httpd* to service requests of the form:

```
http://server-name/~user-name/blah.html
```

by remapping it to the file:

```
~user-name/public_html/blah.html
```

where *user-name* is a user in the */etc/passwd* file. If *~user-name/public_html* does not exist, the person requesting this URL gets an error.

If you specify DISABLE for the `UserDir` directive, the remapping of *~user-name* doesn't occur. The URL above only works if there is a directory called *~user-name* (with the ~) in the `DocumentRoot`.

You should read Chapter 21, *Web: Access Control and Security*, before you enable this Web server feature. You need to understand and minimize potential security risks that your user community might unknowingly create.

18.1.11 Conventions for Public Access

There are several conventions that you should follow as you set up your World Wide Web server for public access. One involves the server name that you have already set up. Although the server name is by default the same as your system name, this is not recommended. The Web convention is that your server name should start with *www*. If your host is *zeus.fredonia.edu*, you might want your server name to be *www.fredonia.edu*. It pays to have a CNAME alias record that maps the Web server name to an actual system name. This give you the flexibility to move your Web server around from system to system as your computing resources change or your system load increases. For more information on how to establish a CNAME alias record, see Section 3.2.

Prior to bringing your Web server online for the public, you should establish an email alias that people can use to send comments, tips on broken pointers, or requests for links. By convention, this alias is called the webmaster alias. This is similar in concept to the postmaster or newsmaster. It is generally a good idea to include a reference to *webmaster@domain-name* on the server home page. Usually the webmaster is the person who looks after the server installation, configuration, and maintenance. This may or may not be the same as the person or people providing the HTML content. For content issues, users usually send email to the person identified as the document owner by his or her signature on the bottom of each HTML page.

18.1.12 Announcing Your Server

Once your Web server is online (and it has some valuable content) you may want to announce your server to the Web community. There is no formal registration process, but there are well-established norms. Your announcement should state what

organization, person, or entity owns the Web server, what type of content it contains, whether it is "ready for use" or "under construction," and the URL of the home page. You can:

- Send the announcement message to the *www-announce@www0.cern.ch* mailing list.
- Post the announcement message to a moderated newsgroup called *comp.infosystems.announce*.
- Send the announcement message to *www-request@info.cern.ch*. This gets your announcement on the CERN World Wide Web Servers list. This is how you get your Web server listed alphabetically by continent, country, and state.
- Send the announcement message to *whats-new@ncsa.uiuc.edu* in HTML format and written in the third person. This is how you get your announcement on the NCSA Mosaic "What's New" page.

In the first three cases, you should make sure that your URL stands out visually from the rest of the message. Most people place it on a line by itself with a blank line before and after. Do not embed it in the announcement text with punctuation. The trailing period (.) in a sentence changes your URL when people with cut-and-paste capabilities grab it and try to visit your new Web server.

Prior to hitting the send key, double-check that you have included the URL of your home page in your announcement. Then check three times that it is correct. It's not convenient to change your announcement once it has escaped onto the Internet.

18.2 Web Server Maintenance

Once your Web server is up and running reliably, it requires very little maintenance. The two most common things you will do periodically is update the HTML and rotate the log files. You will also occasionally have to restructure the underlying disk and directory structure without breaking existing hyperlinks that point into your Web server.

18.2.1 Updating HTML Documents

There is no need to stop or reboot your *httpd* server when you update the HTML documents in the document tree. You simply lay the new or modified HTML documents right on top of the existing online document tree. Some people edit their HTML documents in place while the Web server is online for Internet access. Depending on the availability of disk space, others like to keep two parallel environments, one for offline development and one for online Internet access.

We like to keep an offline copy, with relative URLs (described in the next chapter), that we can work on at our own pace. The relative URLs let us preview and verify from a local disk (without involving a Web server). When we're ready to roll it into

public use, we *tar* up the offline copy and then just lay it back down on top of the online version. Another way to do that, assuming both versions are in the same filesystem, is to just move the directories:

```
% cd /usr/local/etc/httpd
% mv htdocs htdocs.old && mv htdocs.new htdocs
```

The directory name *htdocs* is defined as the default DocumentRoot.

18.2.2 Managing the Log Files

On a daily or weekly basis you should move or archive the log files to keep them from growing too big and to facilitate HTTP log analysis. It's a good idea to automate this process.

If you are running *httpd* under *inetd*, this is very easy. You just move the log files, using *mv*, with no effect on the server. The next client request opens up a new `TransferLog`.

If you are using *httpd* standalone, the *httpd* daemon opens the log files once when it starts up and does not close them until *httpd* is killed. To relocate one of the log files, the following steps are required:

1. Move the log file to a new name with the *mv* command. The *httpd* server will continue to write to the same log file, even though it has a new name.
2. Restart the standalone *httpd* server, as described in Section 18.1.5.2.

Step 2 causes all new *httpd* processes to log to the new log file. You may then do what you wish with the old log file.

18.2.3 Moving Directory Structures

Occasionally you will have to move sections of your document tree to deal with disk space constraints or a changing systems environment. This is where the `Alias` directive can be helpful.

Let's look at an example in which you want to spread your document tree over two disk drives. If your original document tree used the default path, and you have three organizations on your Web server, your physical directory hierarchy might look like this:

```
% ls /usr/local/etc/httpd/htdocs
chemistry/
physics/
biology/
```

To move the Biology department to a new drive mounted as */archive/1*, you would take the following steps:

1. Create a *biology* directory on */archive/1*.
2. Copy the current *biology/* directory tree to */archive/1/biology*.
3. Add an `Alias` directive to your *srm.conf* configuration file:

   ```
   Alias /biology/ /archive/1/biology/
   ```

   Reboot your *httpd* server if it is running standalone so that the new Alias directive takes effect.
4. Delete the current */biology* directory.

Using this technique, URLs of the form *http://www.fredonia.edu/biology/* continue to work transparently, even though the underlying directory structure has been changed.

18.2.4 Mirroring Documents and Servers

The *htget* Perl script gets HTML non-interactively from a Web server. You can use it to get a single document, or to mirror (duplicate) part or all of another host's directory tree. Why would you want to do that?

- Are you making heavy use of a remote server, with lots of graphics or over a slow line? Pre-fetching the HTML document and the graphics in it—in the background, or overnight when the network is faster—will speed up your server.
- If you have a heavily used server, other servers can mirror your server's HTML documents and spread out the load.
- You want a quick and easy way to grab an HTML document from the UNIX command line or from a non-interactive program like a shell script or a *cron* job. *htget* can write the document to a file or to its standard output.

Oscar Nierstrasz wrote *htget*. You can get it from this book's archive (see the Preface). The version at *http://cui_www.unige.ch/ftp/PUBLIC/oscar/scripts/README.html* may have been updated since we grabbed it.

htget has several options. These make big changes in what it does, so it's important to decide which one you need:

- With no option, *htget* copies the HTML file you specify to your current directory:

  ```
  % htget http://nearnet.gnn.com/mkt/travel/center.html
  htget: created center.html (2203 bytes)
  ```

- The *–s* option writes the file to standard output. For example, you could search an HTML document by typing:

```
% htget -s http://nearnet.gnn.com/mkt/travel/center.html | grep Thailand
```

- The *–abs* option converts all relative URLs to absolute URLs. If you're retrieving a single file that has relative links in it, this option makes sure that you can access all the links from your local copy.

  The following are two examples of the same HTML document, *http://tbl.com/tbl/TBLhome.html*. These lines were copied with no *htget* options:

  ```
  <IMG ALT="TBL" SRC="graphics/DIRECTORY.gif">
  <A HREF="/tbl/tbl.html">
    <IMG SRC="/tbl/graphics/whatsup.gif" ALT="[What's Up at TBL]"></A>
  ```

  Here are the same lines with *–abs*:

  ```
  <IMG ALT="TBL" SRC="http://tbl.com:80/tbl/graphics/DIRECTORY.gif">
  <A HREF="http://tbl.com:80/tbl/tbl.html">
    <IMG SRC="http://tbl.com:80/tbl/graphics/whatsup.gif"
    ALT="[What's Up at TBL]"></A>
  ```

- Finally, the *–r* option copies recursively. *htget –r* creates subdirectories under your current directory, and also retrieves graphics and other files. This re-creates the remote tree in a way that lets you use it locally (either directly within a Web browser or in a server document tree). The script uses some interesting tricks to make your local copy of the tree correct:

  – It gets the file and all files that are reachable from that URL, but only when the files are in the same directory hierarchy on the same host. This avoids copying a much larger tree than you expected.

  – When *htget* retrieves a file, any absolute URLs in other retrieved documents that point to that file are converted to relative URLs. This makes sure that you use your local copy of a file whenever possible.

  – When *htget* doesn't retrieve a file (because it's from another tree), any relative URLs in other retrieved documents that point to that file are converted to absolute URLs (including the remote host name and port number). This makes sure that you can always access files that weren't copied.

htget is only good for getting `http:` URLs. There are "hooks" in the code for `ftp:` and `gopher:` URLs, but the functionality isn't there.

Before you start grabbing and using all the HTML files you can find `:-)`, remember the copyright implications of copying other people's work. You shouldn't grab people's stuff without asking, even if they are giving it away.

18.2.5 HTTP Logs and Log Analysis

There are two Web server log files, the `Error` log and the `Transfer` log. By default they are in *logs/error_log* and *logs/access_log* in the `ServerRoot` directory. You can change

the location or filename with the ErrorLog and TransferLog directives in *httpd.conf* (see Appendix E, *Web: httpd.conf Directives*.

After you have announced your Web server, you will want to know who is using it and what for. There are several readily available tools to help you analyze activity on your Web server. Every time someone grabs a file using your Web server, *httpd* logs the host name, the date and time, and the URL request. A typical log file looks like:

```
topcat.ug.eds.com - - [29/Jul/1994:04:07:11 -0700] "GET /Update/940620.html ▶▶▶
▶▶▶HTTP/1.0" 200 4327
cathy.ijs.si - - [29/Jul/1994:04:07:19 -0700] "GET/info.home.html ▶▶▶
▶▶▶HTTP/1.0" 200 2241
cathy.ijs.si - - [29/Jul/1994:04:07:45 -0700] "GET/master-subject.html ▶▶▶
▶▶▶HTTP/1.0" 200 9348
topcat.ug.eds.com - - [29/Jul/1994:04:07:47 -0700] "GET/whats-new.html ▶▶▶
▶▶▶HTTP/1.0" 200 28123
144.122.171.70 - - [29/Jul/1994:04:08:36 -0700] "GET / HTTP/1.0" 200 2153
cathy.ijs.si - - [29/Jul/1994:04:08:40 -0700] "GET /key-vax-index.html ▶▶▶
▶▶▶HTTP/1.0" 200 77847
ceng.metu.edu.tr - - [29/Jul/1994:04:08:55 -0700] "GET/pics/dec-logo.gif ▶▶▶
▶▶▶HTTP/1.0" 200 15 36
silver.fe.msk.ru - - [29/Jul/1994:04:10:06 -0700] "GET /demo.html HTTP/1.0" 200 786
```

This log file is in what is called the common log format because most Web servers generate it. Formally, the format of your access log is:

```
host rfc931 authuser date-time request status bytes
```

Where each item means:

host
: Hostname

RFC931
: The RFC 931 username if active, a dash (–) if not. It is normally (–) if *httpd* is configured with IdentityCheck off, which is the default. If IdentityCheck is turned on, *httpd* attempts to use the RFC 931 protocol to talk to an RFC 931-compliant daemon on the system running the Web browser to determine the name of the user running the Web browser. This whole thing hinges on whether users' systems are configured with an Identity daemon. Most are not.

authuser
: HTTP/1.0 authenticated user, a dash (–) if none. It is normally (–) unless the directory containing the file being accessed is protected with user authentication. If so, authuser is the authorized username that was used to access the file.

date-time
: Local time of the request, with time zone offset from GMT at the end

request
: The request as sent by the client

status
: The HTTP/1.0 status code from this transaction

bytes
: The number of bytes sent in this transaction, not including the header. If not applicable, this is a dash (–).

The common log format is supported by the latest CERN and Plexus Web servers. Many Gopher servers also support it. This helps in the development of log analysis tools like *getstats*, *wwwstat*, and *wusage*. These tools are not included as part of the NCSA HTTP kit, so you will have to go grab one yourself. Here is a summary of some of these tools:

18.2.5.1 *getstats*

getstats is a C program that produces Web server statistics on everything from a monthly basis down to an hourly basis. You can cut the logs by domain, page request, or directory tree. It works with any Web server supporting the common log format. There is even an HTML forms interface for *getstats*.

Just point your Web browser at:

```
http://www.eit.com/software/getstats/getstats.html
```

Author: *getstats* was developed by Kevin Hughes, *kevinh@eit.com* at Enterprise Information Technology.

18.2.5.2 *wwwstat*

Author: *wwwstat* is written in Perl. *wwwstat* does not make any changes to or write any files in the server directories. This means that you can assign the log analysis to someone without root privileges.

Just point your Web browser at:

```
http://www.ics.uci.edu/WebSoft/wwwstat/
```

wwwstat was developed by Roy Fielding, *fielding@ics.uci.edu* at the University of California, Irvine, Department of Information and Computer Science. To see an example of what *wwwstat* can do, point your Web browser at:

```
http://www.ics.uci.edu/Admin/wwwstats.html
```

wwwstat includes a companion tool called *gwstat* that can translate *wwwstat* summary data into graphical charts. Figure 18-1 shows the results of *wwwstat* and *gwstat* used together to summarize the hourly traffic on a Web server. Because the graphical summaries are in GIF format, they can be included with HTML.

18.2.5.3 *wusage*

Author: *wusage* is a C program that produces Web server statistics on a weekly basis and produces graphs of the results in GIF format. Pie charts by domain show the geographic distribution of users. A graph of accesses over time shows server usage growth; a smaller version this graph is provided for use as a home-page icon, linked to the statistics page. Textual versions of the above information, in addition to the top ten sites and top ten URLs requested, are output for each week in HTML format.

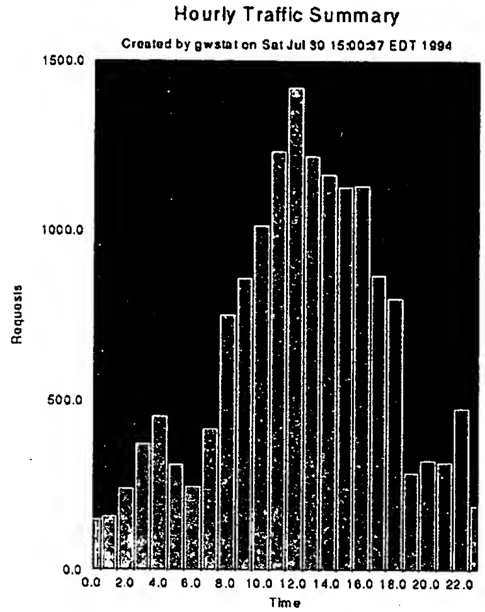


Figure 18-1. gwstat hourly traffic summary

Just point your Web browser at:

```
http://siva.cshl.org/wusage.html
```

wusage was developed by Thomas Boutell, *boutell@netcom.com* at Cold Spring Harbor Labs, Cold Spring Harbor, New York. Figure 18-2 shows an example of what *wusage* can do.

18.3 Enabling More Features

Once you've got your basic Web server operating, there are a few useful features you can enable if you want to. You can add support for new MIME types or expand how the Web server recognizes existing types. You can enable server-side includes, which allow the Web server to automatically fill in the last-modified date or read a file into an HTML file before sending it to the user. Or you can customize how the Web server displays directories.

18.3.1 Adding MIME Types

As your server matures, you may find file types that aren't supported. As described earlier, Web browsers and Web servers using HTTP/1.0 pass MIME types back and forth.

You can add support for new MIME types in your server (or new ways for the server to recognize existing types).

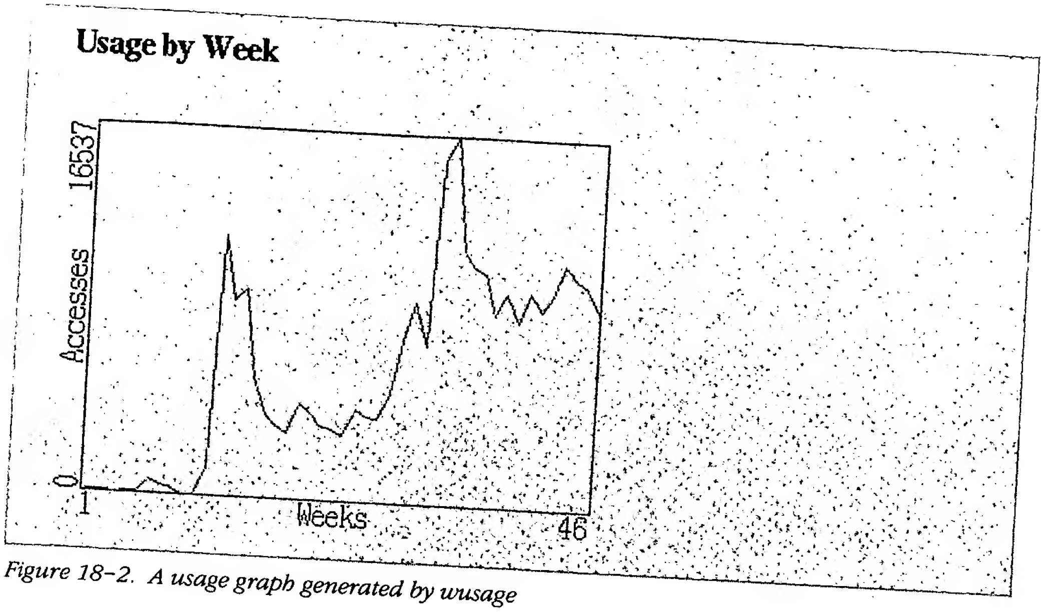


Figure 18-2. A usage graph generated by wusage

You should be aware that adding new types may also require users to add support on their end. On the client side, Web browsers that are HTTP V1.0-compliant know how to use the MIME type that comes back from the Web server. They use the *.mailcap* configuration file to decide which external players and viewers to invoke for each incoming data type.

In typical use, if a user retrieves a file that ends with the *.ps* file extension, the Web server passes the file back to the Web browser and tells the Web browser using MIME that it is a PostScript file. The Web browser caches the PostScript file to a local temporary directory, then looks up the PostScript viewer command from the *.mailcap* file. It then invokes the PostScript viewer and hands the viewer the local copy of the PostScript file. This same technique is used with motion picture files, audio files, and some types of images.

On the server side, the Web server knows how to map file types and file extensions to standard MIME data types. This is controlled in the *mime.types* configuration file. This configuration file is very complete for most people. However, there are directives that you can add to the *srm.conf* configuration file to add special MIME extensions to your Web server:

`AddType`
: Allows you to override MIME-type definitions found in the *mime.types* configuration file

`AddEncoding`
: Adds unique MIME encoding directives for the client/server handshake

`DefaultType`
: Used to establish a default MIME-type to be returned to the Web browser

Full definitions of these directives can be found in Appendix F, *Web: srm.conf Directives.*

A good example of the use of the `AddType` directives is to extend *httpd*'s ability to identify HTML documents. *httpd* determines that a file contains HTML by checking to see if it meets one of these two conditions:

- The first line of the file contains an <HTML> tag
- The filename ends with a *.html* file extension

The *.html* file extension is used because the default *mime.types* configuration file identifies:

```
text/html html
```

If you were serving HTML documents that had been authored on an MS-DOS personal computer, they would typically end with a *.htm* file extension, due to the 8.3 filename size restriction. You can tell *httpd* that all files ending with a *.htm* file extension should be typed as HTML by adding another MIME type definition:

```
AddType text/html htm
```

You can also use MIME type information to extend the list of document types that Web browsers know how to process. For example, by adding MIME type information for WordPerfect documents, you could serve WordPerfect documents (in their proprietary format) off your Web server and configure Windows-based Web browsers to invoke WordPerfect when WordPerfect documents are retrieved via URL.

18.3.2 Enabling Server-Side Includes

Server-side includes are similar to include files in any programming language; however, they can include not only files, but also the values of environment variables. They are an easy way to consistently include dates, document size, author names, or any HTML in a number of HTML documents. For an example of a server-side include, and to see how it is used in HTML, see Section 19.1.10. See Section 21.2.5 for more information about the security implications of server-side includes.

Server-side includes are not enabled by default in the NCSA *httpd* server. Server-side includes can be enabled server-wide or on a per-directory basis by specifying the `Includes` or `IncludesNoExec` clause on the `Options` directive in *access.conf* or *.htaccess* (see Chapter 21, *Web: Access Control and Security*, for more on these files):

```
Options IncludesNoExec
```

The `IncludesNoExec` option is more secure than Includes (for reasons described in Section 21.2.5).

You also need to specify which files get the includes inserted. How you do that depends on how many files you want processed. If you plan to use server-side includes in almost every file, add a single directive to the *srm.conf* configuration file:

```
AddType text/x-server-parsed-html .html
```

Using server-side includes does take CPU cycles and you might not want to do that on every file. If you plan to use server-side includes sparingly, you should (instead of the above) define a new file extension to identify the files that need processing. Modify the *srm.conf* configuration file to add:

```
AddType text/html shtml
AddType text/x-server-parsed-html .shtml
```

The convention is to use *.shtml* as the file extension for files that need server-side include processing. The first directive tells *httpd* that files with the *.shtml* extension are to be treated as HTML documents. The second directive tells *httpd* that only files that have the *.shtml* file extension are processed for possible server-side includes. You'll have to remember which files are *.shtml* files instead of *.html* files when creating links to these files.

18.3.3 Automatic Directory Indexing

When a URL points to a directory instead of a file, the Web server first tries to return the file *index.html* in the specified directory.* If *index.html* doesn't exist, then the Web server can automatically generate HTML text describing the contents of the directory. This is called automatic directory indexing. Figure 18-3 shows an automatically generated directory index displayed with Mosaic.

You can enable or disable automatic directory indexing in the global ACF (*access.conf*) (or the per-directory ACFs) using the `Options` directive. By default, it is enabled server-wide. When it is disabled for a directory, and there is no *index.html* in that directory, the Web server returns an error whenever a user tries to use that URL.

There are two types of automatically generated directory index, plain or fancy. The type that's used can be controlled on a server-wide or per-directory basis in *srm.conf*. The initial *srm.conf* configuration file has `FancyIndexing` enabled server-wide.

With plain indexing, the filenames are listed in a single column, similar to the *ls -1* (that's the number one, not lowercase L) shell command. The user can click on the filename to retrieve the file.

With fancy indexing, an associated icon, the file size, and an optional file type description are displayed next to each filename.

You can control the icon associated with each file type. For example, sound files (identified by their MIME type of audio/) by default have a sound icon. You probably wouldn't want to change it, but you could define a different sound icon, or an icon for some type that has no default icon.

When an icon is associated with a filename, a reference to that icon is included as an inline graphic in the HTML that the Web server automatically generates. The icons are

* Or the filename identified by the *DirectoryIndex* directive in the *srm.conf* configuration file.

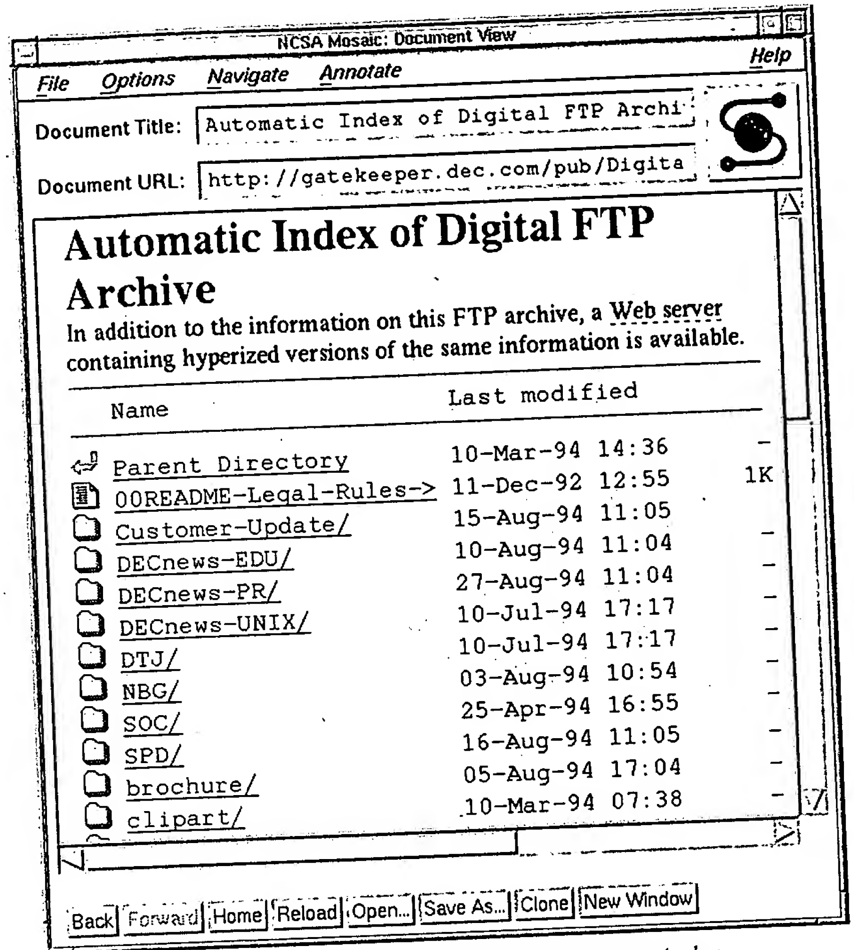


Figure 18-3. Mosaic screen shot of an automatic directory index

neither part of nor embedded in Web browsers. Instead, they are just like any other image that you can serve from your Web server.

The optional file type description of fancy directory indexing is useful in non-graphical Web browsers.

Fancy indexing also allows you to control header and trailer information displayed before and after a directory index. The header information, by default the *HEADER* file, is displayed prior to the actual directory index. The trailer information, by default the *README* file, is displayed after the directory index. This header and trailer information can be plain text or HTML. You can also tell the Web server (using the `IndexIgnore` directive) to ignore certain filenames when generating an automatic directory index. By default *HEADER* and *README* are ignored, so they don't appear in automatic directory listings.

Automatic directory indexing is useful if your Web document tree is the same as your FTP archive directory hierarchy. Web browsers can navigate the directory hierarchy of your FTP archive and retrieve files by *ftp*. With fancy indexing turned on, you can visually enrich the information on your FTP archive even though there are no HTML documents contained in that FTP archive. But this enrichment only works when the FTP archive is viewed through your Web server—not if a Web client uses FTP to go directly to the FTP archive.

Automatic directory indexing is controlled with directives in the *srm.conf* configuration file. Here's a quick summary of the relevant directives:

`AddIconType` Defines which icons are associated with documents according to MIME-type mapping

`AddIcon` Defines which icons are associated with documents according to file types and names

`DefaultIcon` Defines which icon to use when no icon can be automatically associated with a given file

`AddDescription` Defines the descriptive phrase associated with a particular type of file

`IndexOptions` Defines whether icons are treated as hyperlinks, whether HTML documents are scanned for document titles to be used as descriptive text phrases, and the degree to which file characteristics are suppressed

For a reference to each *srm.conf* directive, see Appendix F, *Web: srm.conf Directives.*

The defaults for automatic directory indexing are:

```
AddIconByType (TXT,/icons/text.xbm) text/*
AddIconByType (IMG,/icons/image.xbm) image/*
AddIconByType (SND,/icons/sound.xbm) audio/*
AddIcon /icons/movie.xbm .mpg .qt
AddIcon /icons/binary.xbm .bin
AddIcon /icons/back.xbm ..
AddIcon /icons/menu.xbm ^^DIRECTORY^^
AddIcon /icons/blank.xbm ^^BLANKICON^^
DefaultIcon /icons/unknown.xbm
ReadmeName README
HeaderName HEADER
IndexIgnore */.??* *~ *# */HEADER* */README*
```

The `AddIcon` line that uses `^^DIRECTORY^^` defines the icon used to indicate directories in a directory listing. The `^^BLANKICON^^` icon is used just to fill in space in the directory listing output, so that everything else lines up like a table when the browser processes the HTML.

18.4 Installation Summary

At this point you have a working Web server online. You have pulled the NCSA Web server kit, configured it for your site, moved it into place, started it up, and tested it. You have a home page and, with enough content to justify it, have announced it to the online Web community. People are accessing it, and you are generating logs and analyzing the log files to determine usage and trends.

Congratulations—you are now a *webmaster*!

In this Chapter:
- *HTML Overview*
- *Information Design Issues*
- *HTML Authoring Tools*
- *Clickable Image Maps*

19

Authoring for the Web

Anyone can create documents for the Web. You don't need to know anything about server administration. It is common for the people developing, or authoring, Web documents to be different from the person installing and running the Web server.

Web documents are written in HTML, the HyperText Markup Language. Most Web browsers can read HTML directly from a file, without the involvement of a server. That means that you can write your HTML and test it without making the document available to the public during the development and without setting up a separate server.

This chapter describes the HTML language, including how to create links to other documents and services. Near the end of the chapter, we describe some of the tools available to make HTML authoring easier. This chapter also describes how to create clickable image maps, which allow you to create links from certain areas in an image instead of from certain words in a document.

19.1 HTML Overview

An HTML document is an ASCII text file that contains embedded HTML tags. On a UNIX server, it typically has a filename extension of *.html*. In general, the HTML tags are used to identify the structure of the document and to identify hyperlinks (to be highlighted) and their associated URLs.

HTML identifies the structure of the document and it suggests the layout of the document. The display capabilities of the Web browser determine the appearance of the HTML document on the screeen.

Using HTML you can identify:

- The title of the document
- The hierarchical structure of the document with header levels and section names
- Bulleted, numbered, and nested lists

- Insertion points for graphics
- Special emphasis for key words or phrases
- Preformatted areas of the document
- Hyperlinks and associated URLs

HTML cannot control the:

- Typeface used for any document component
- Point size of any specific font
- Width or height of the screen
- Centering, spacing, or line breaks of information, except in preformatted text
- Background, foreground, or highlight colors

These things all depend on the browser, which may allow the user to control them.

19.1.1 Autoflowing and Autowrapping

The most basic element in the HTML document is the paragraph. The Web browser flows all the contents of the paragraph together from left to right and from top to bottom given the current window or display size. This is called *autoflowing*. How you break lines in that paragraph in the HTML is irrelevant when that page is displayed by a Web browser.

The Web browser wraps anything that doesn't fit on the current line, putting it on the next line. For example, a paragraph that displays six lines long on an 8-inch wide window rewraps to be about 12 lines long if the user resizes the Web browser window to be half as wide. This is called **autowrapping**.

Your document will be read by both graphical and character-based Web browsers. Furthermore, there will be display differences with graphical Web browsers given different screen resolutions. So just because one browser breaks a line at one place, that doesn't mean others will do so at the same place. Just remember that on the Web, you live in a world that is left-justified and flows from top to bottom.

19.1.2 HTML Tag Syntax

HTML tags are encapsulated within less-than (<) and greater-than (>) brackets. Some of the tags are single-element tags that can stand by themselves. These are referred to as standalone tags. The syntax is simple:

```
<tag>
```

The most common standalone tag is <P>, which ends a paragraph.

Other tags are used in pairs. The beginning tag tells the Web browser to start the tag function and the ending tag tells the Web browser to stop. The ending tag is created by adding a forward slash (/) to the beginning tag. The syntax is:

```
<tag>object</tag>
```

The tag identifies the function that is being applied to the object. For example, if you wanted to add special emphasis to a phrase, you would encapsulate the phrase with the <EM> tagging pair as illustrated:

```
<EM>text to emphasize</EM>
```

Many of the standalone tags and the beginning tag of tagging pairs can have options included. So to be complete the syntax is:

```
<tag option1 option2 option3>
```

A mistake you will make when manually authoring HTML with a text editor is to forget the ending tag or leave out the slash (/) in the ending tag. Don't worry—it will be obvious when this happens, and it's simple to correct.

19.1.3 Document Construction Guidelines

Now let's look at the three tagging pairs used to create the highest level of structure in an HTML document:

```
<HTML> entire HTML document </HTML>
<HEAD> document header information </HEAD>
<BODY> body of the HTML document </BODY>
```

The following is a skeletal HTML document that shows the required nesting of these three tagging pairs:

```
<HTML>
        <HEAD>
                Head elements
        </HEAD>
        <BODY>
                Body elements and content
        </BODY>
</HTML>
```

Remember that physical layout in an HTML document, like indentation and line breaks, is meaningless to Web browsers. So you can format your HTML according to your own preferences.

19.1.4 Sample HTML Document

The following is an example of the HTML template that I keep in my home directory. I pull it into a text editor as a starting point for creating a new HTML document.

Although you could have a slightly more bare-bones HTML template, it's easier to delete than add. Here is the sample HTML document:

```
<HTML>
<HEAD>
<TITLE>Sample HTML Document</TITLE>
<LINK REV="OWNER" HREF="mailto:rjones@ix.netcom.com">
</HEAD>
<BODY>
<IMG SRC="my-logo.gif" ALT="logo">
<H1>Sample HTML Document</H1>
<EM>To demonstrate HTML style</EM>
<P>
<HR>
<P>
Hello World.
<P>
<HR>
<P>
Creation Date: <EM>Sat Apr 9, 1994</EM>
<P>
<ADDRESS><A HREF="rjones.html">DRJ</A></ADDRESS>
</BODY>
</HTML>
```

Figure 19-1 shows how this sample HTML document looks when displayed with Mosaic.

Let's go through it and look at the new tags it uses:

`<TITLE> text-phrase </TITLE>`
: The <TITLE> tag pair is used once per document and identifies the document title. It should appear at the top of the document within the <HEAD> section. Many Web browsers display the document title in a special area. This is also the document name that is used when a user adds the document to her hotlist or list of bookmarks for later reference. Although not absolutely required, it is a good idea to have a document title in each HTML document.

`<LINK REV="OWNER" HREF="internet-address">`
: The <LINK> standalone tag is used in a special way here to identify the email address of the document owner. This information is not displayed by a Web browser, but is used by some browsers when sending an email message on behalf of the user to the document owner. Other uses of the LINK tag will be described later.

`<IMG SRC="URL">`
: The <IMG> standalone HTML tag inserts an image into the current autoflow stream. Note that you specify the image with its URL. See Section 19.1.7 for more on URLs, and Section 19.1.9 for more on image insertion.

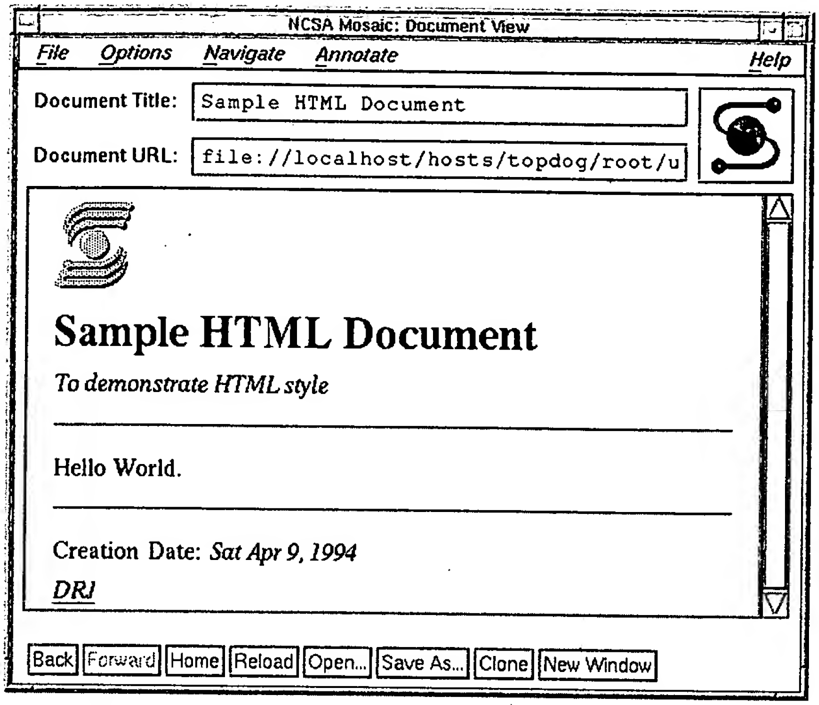


Figure 19–1. Sample HTML document display by Mosaic

<H1>*text-phrase*</H1>
: The <H1> header-level tag pair indicates a level of structure in your document. There are six levels of headers, H1 through H6. Headers, like paragraphs, autowrap if the phrase cannot fit on a single line. Header tags also force paragraph breaks before and after the header.

<EM>*text*</EM>
: The <EM> tagging pair marks a phrase to be rendered with generic emphasis. On a graphical display, the phrase could be italicized, while on a character-mode display it might be underlined.

<P>
: The <P> tag ends a paragraph and places a blank line after the paragraph on the display. Two paragraph break tags result in a single blank line.

<HR>
: The <HR> (horizontal rule) tag forces the Browser to generate a horizontal rule, or line, across the display. It breaks pages into logical sections and is useful when creating forms. There is no equivalent vertical rule.

<ADDRESS>*text*</ADDRESS>
: The <ADDRESS> tagging pair is used to identify addresses. Depending on the Web browser, the text is rendered in a special point size, typeface, or font.

19.1.5 Hyperlinks

One of the most important HTML tagging pairs creates a hyperlink to another document or Internet resource. Generically, the Anchor tagging pair is used as follows:

```
<A option1 optionN>anchor-text</A>
```

It's called an Anchor tagging pair because this tag anchors the link to a particular spot in your HTML. Another use of the Anchor tag described later marks a point within a document where a link transfers to (it is optional, because by default a link transfers to the beginning of a document). The two variations are sometimes called **link-from** and **link-to** tags.

The beginning Anchor tag always requires at least one option. Although there are several options, the two most important are HREF, which defines a hyperlink, and NAME, which identifies a link-to destination within a file. Let's look at the HREF option first. Here's the syntax:

```
<A HREF="URL">anchor-text</A>
```

The *anchor-text* is displayed in the browser. When the user triggers the link, the browser retrieves the specified URL. URLs are described in detail in Section 19.1.7.

Going back to our sample HTML document, there is a hyperlink at the bottom of the document:

```
<A HREF="rjones.html">DRJ</A>
```

I use this hyperlink to sign each of my pages. My initials, DRJ, become the anchor text for a hyperlink to another HTML document called *rjones.html* in the same directory.

My signature hyperlink is actually enclosed within an <ADDRESS> tagging pair:

```
<ADDRESS><A HREF="rjones.html">DRJ</A></ADDRESS>
```

This demonstrates how hyperlinks can be embedded in other tagging constructs. Anywhere you have text within the body of the HTML document, you can create a hyperlink. The text could be in a paragraph, a header, a quote, or as part of an address. However, the reverse is not always true. You can put an anchor tagging pair inside a header, but you should not embed headers inside an anchor pair.

Instead of using text as the hyperlink anchor, you can use an image. This is done by enclosing the <IMG> tag with the anchor tagging pair:

```
<A HREF="org-overview.html"><IMG SRC="my-logo.gif"></A>
```

When the inline image is displayed, it becomes sensitive. If the user triggers it, the browser retrieves the document identified by the associated URL, in this case a file called *org-overview.html*.

19.1.6 Linking to Points Within Documents

By default, links point to the top of a document. You can also create hyperlinks that jump to other points in an HTML document. First, you must identify and name the link-to point. This is done with the NAME option of the Anchor tag.

```
<A NAME="anchor-name">anchor-text</A>
```

This tagging pair identifies a link-to point in the HTML document and names that point. The tagging pair provides an alternative entry point into an HTML document. The anchor-text cannot be omitted.

This link-to point is then referenced by using *#anchor-name*. For example, say you have a document with an appendix in the same file, and you want to link to the appendix. The header that starts the appendix should be turned into a link-to point. The HTML that indentifies the beginning of your appendix might look like:

```
<H2><A NAME="Appendix-A">Appendix A</A></H2>
```

You can then create hyperlinks to the appendix from within the same document like this:

```
Details are in <A HREF="#Appendix-A">Appendix A</A>.
```

You could also link directly to the appendix from an external document. From another HTML document you would reference it as:

```
<A HREF="URL#Appendix-A">Appendix A</A>
```

The URL can be a fully qualified URL or a relative URL.

If the syntax of the anchor is incorrect, most Web browsers are forgiving and display the anchor-text or inserted image, but not make it sensitive as a hyperlink. This can be done intentionally to temporarily disable links without removing all the HTML that defines the link.

19.1.7 URLs

A URL points to a file or directory.* There are three distinct types of URLs: absolute, relative, and local. There are also a couple of special-case URLs supported by some browsers.

Absolute URLs completely describe how to get a file on the Internet. Relative URLs and local URLs are both ways to specify a file on the same server as the document they appear in. The following sections describe each type of URL in more detail.

* The file may be a script or a program, rather than a document. That's described in Chapter 20, *Web: Gateways and Forms.*

19.1.7.1 Absolute URLs

Absolute URLs can be used to reference any resource on the Internet, including local resources. However, it's better to use relative URLs for local resources for reasons described in the next section. The full syntax for an absolute URL is:

```
access-method://server-name[:port]/directory/file
```

If no port is specified, the default port associated with the given access method is used. The URL of a document that is on a Web server has an access method of *http:*. For example, the master list of all public World Wide Web servers throughout the world can be accessed using the following URL:

```
http://info.cern.ch/hypertext/DataSources/WWW/Geographical.html
```

This means that by using the HTTP protocol to reach a server called *info.cern.ch* (using the default port), there is a directory */hypertext/DataSources/WWW* that contains a hypertext document named *Geographical.html*. Every file on the Internet is uniquely addressable by its URL, regardless of the type of file or the type of server it is provided by.

Besides the *http:* access method, the URL concept also supports other important Internet protocol access methods such as Gopher, FTP, and Telnet.

The URL to access a file on a Gopher server follows the standard remote URL specification. Here's the URL to the campus Gopher server at the University of Minnesota:

```
gopher://gopher.micro.umn.edu/
```

To look at the FTP archive run by O'Reilly & Associates, you could open the following URL:

```
ftp://ftp.ora.com/pub/
```

You might sometimes see references to both the *ftp:* and the *file:* access methods. The *ftp:* nomenclature has replaced the earlier *file:* nomenclature, as it more accurately reflects the protocol being used. When using the *ftp:* access method to reference the contents of a directory you should make sure you close the URL with a trailing slash (/). When referencing a file in a directory, this is not necessary.

The URL to a Telnet resource is no surprise:

```
telnet://info.cern.ch/
```

When any URL appears in one of your documents, the Web browser uses the URL to go directly to that resource; it doesn't go through your Web server. That means that users who cannot *telnet* directly to a host from the shell command line (because they are blocked by a security firewall or because they do not have the password) will not be able to access the host via a Web browser either, even if you can access it from the machine where your HTML resides.

If any server is running on an unusual port, you need to include the port number as part of the URL. The standard ports are:

Port	Service
21	FTP
23	Telnet
70	Gopher
80	HTTP

The following URL shows how to specify a Gopher server running on port 1250:

```
gopher://garnet.berkeley.edu:1250/
```

Many sites around the Web have created public-access gateways to other common Internet services. The URLs for these gateways are based on file URLs, but can have various special syntaxes. Section 20.3 describes several WAIS gateways.

19.1.7.2 Absolute versus relative URLs

A relative URL assumes the same access-method, server-name, and directory-path as the document the URL appears in. It indicates the relative position of the target URL from the current URL. For example, here's a URL to a file in the same directory as the document the URL appears in.

```
<A HREF="page-7.html">Next Page</A>
```

Relative URLs are also sometimes called partial URLs. They are used to point to information resources in the same directory or on the same server. Absolute URLs, on the other hand, are usually used to point to information resources on other servers. In practical terms, this means that you can use relative URLs to direct navigation between documents that you author and use absolute URLs to direct navigation to resources elsewhere on the Internet.

The distinction between absolute and relative URLs is totally hidden from the Web server. When a user selects a relative hyperlink, the Web browser uses the current URL to determine the access-method, server-name, port-number, and directory-path, and sends only absolute URLs to the server.

Relative URLs allow you to author your HTML with total disregard for where your final HTML directory structure will be placed on the Web server or even what system the server will run on. You can direct the relative navigation between HTML documents using the same directory path navigation constructs that you use on UNIX to move up and down a directory tree. For example, to have a hyperlink that jumps up two levels in the directory structure to your home page, the following HTML could be used:

```
<A HREF="../../home.html">Return To Home Page</A>
```

Relative URLs also give you the flexibility to move your HTML directory structure anywhere on the Web. Your administrator may want another site to have a local copy of your HTML directory structure for various reasons.

Relative URLs also work when files are read directly from the file system by a Web browser, without the intervention of a Web server. This is very useful for Web authoring.

Don't use symbolic links in your document tree, because they defeat the purpose of relative URLs, and make the document tree unportable. If you are tempted to use a symbolic link, use the Alias directive in *srm.conf* instead. (This is described in more detail in Section 18.2.3.)

A relative URL can be relative to the document root instead of the directory of the current document. This kind of relative URL starts with a slash. In other words, it looks like an absolute URL, but without the access method and server name.

19.1.7.3 Oddball URLs

There are a couple of other URLs supported by certain browsers. They access articles from the local news server or send an email message to a document's owner.

For example, to read the *comp.infosystems.www.providers* newsgroup with your Web browser instead of a news reader, you would open the following URL:

```
news:comp.infosystems.www.providers
```

Some Web browsers, such as Lynx, support sending email back to the owner of a page. To identify ownership of a page, you can use the following local URL, as shown in the sample HTML document in Section 19.1.4:

```
mailto:internet-address
```

However, `mailto:` is not widely supported yet.

19.1.8 Lists

Another key HTML construct is the list. There are three types of lists:

- Unordered (bulleted) List: <UL>list</UL>
- Ordered (numbered) List: <OL>list</OL>
- Definition List: <DL>list</DL>

In the first two types of lists, the elements of the list are designated with the <LI> list-item tag. For example, a simple bulleted list with three items would be tagged like this:

```
<UL>
<LI>First bullet
<LI>Second bullet
```

```
<LI>Third bullet
</UL>
```

As you might expect, if a list-item phrase is wider than the display width, the Web browser autowraps the phrase and aligns the next line to indent under the previous line. Lists can also be nested:

```
<UL><LI>First bullet
        <UL><LI>First sub-bullet
        <LI>Second sub-bullet
        <LI>Third sub-bullet</UL>
<LI>Second bullet
<LI>Third bullet</UL>
```

Figure 19-2 shows how these nested lists would be rendered by Mosaic.

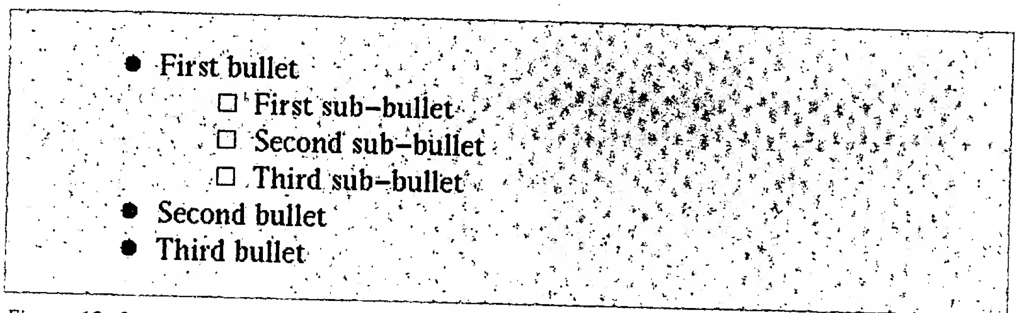


Figure 19-2. Nested bullet list rendered with Mosaic

The third type of list, the definition list, is used when the list items do not need to be bulleted or numbered and can stand on their own. A good example might be a glossary, which lists terms and their definitions. Definitions do not use the <LI> tag to mark a new list entry. Instead they use the <DT> and <DD> tagging pairs. The <DT> tagging pair identifies the primary text and the <DD> tagging pair identifies the text associated with and indented below the <DT> tagging phrase.

For example, a definition list of three publications looks like:

```
<DL>
<DT>Action Plan for African Primate Conservation</DT>
<DD>Compiled by J.F. Oates and the IUCN/SSC Primate Specialist Group,
1986, 41 pp.</DD><P>
<DT>Dolphins, Porpoises and Whales. An Action Plan for the
Conservation of Biological Diversity</DT>
<DD>Second Edition. Compiled by W.F. Perrin and the IUCN/SSC
Cetacean Specialist Group, 1989, 27 pp.</DD><P>
<DT>Otters. An Action Plan for their Conservation</DT>
<DD>Compiled by P. Foster-Turley, S. Macdonald, C. Mason and the
IUCN/SSC Otter Specialist Group, 1990, 126 pp.</DD>
</DL>
```

Figure 19-3 shows how this definition list would be rendered by Mosaic.

Action Plan for African Primate Conservation
    Compiled by J.F. Oates and the IUCN/SSC Primate Specialist Group, 1986, 41 pp.

Dolphins, Porpoises and Whales. An Action Plan for the Conservation of Biological Diversity
    Second Edition. Compiled by W.F. Perrin and the IUCN/SSC Cetacean Specialist Group, 1989, 27 pp.

Otters: An Action Plan for their Conservation
    Compiled by P. Foster-Turley, S. Macdonald, C. Mason and the IUCN/SSC Otter Specialist Group, 1990, 126 pp.

Figure 19-3. Definition list rendered with Mosaic

19.1.9 Graphics

Graphics were briefly mentioned earlier when the <IMG> HTML tag was introduced. Fundamentally, there are two different ways to present graphics—inline images and external images. Inline images are displayed by the Web browser as part of your document and are automatically retrieved along with the document. External images are displayed by a separate viewer program (started by the Web browser when needed) and must be specifically requested by triggering a hyperlink.

Inline graphics involve the transfer of a lot of data and so retrieving them can be slow. Fortunately, many Web browsers let users optionally delay downloading of the inline images. With delayed downloading, inline images must be triggered by a hyperlink before they will be downloaded. When triggered, they still appear inline, not in an external viewer.

When contemplating the use of graphics within your HTML, you should consider the user community that will be accessing your Web server. Will they be using graphics-capable Web browsers, such as Mosaic, or will they be using line-oriented Web browsers, such as Lynx? The decision is easier if one class of Web browser dominates, but is more complicated if you expect people with both types of browsers to be using your Web server.

19.1.9.1 Inline images

The inline image is just one more piece of information that is included in the autoflow and autowrap of your HTML on the Web browser screen. In other words, an image is treated just like a word. So an image can appear in the middle of a paragraph. If you want the inline image to stand alone you need to make sure you place <P> or
 HTML tags around it in your HTML document.

For inline images, the two supported graphics types are the Graphics Image Format (GIF) and the X Bitmap (XBM) format. GIF images support 256 colors and are the more common of the two image types. X Bitmaps are black and white.

When inline images are autoflowing as part of a paragraph, you can explicitly control the alignment of the image with the text line by using the optional ALIGN option of the <IMG> tag. The three values for ALIGN are:

```
<IMG ALIGN=TOP SRC="filename.GIF">

<IMG ALIGN=MIDDLE SRC="filename.GIF">

<IMG ALIGN=BOTTOM SRC="filename.GIF">
```

TOP alignment places the top of the image even with the top of the current line of text, and so on. If ALIGN is omitted, bottom alignment is the default.

In terms of image placement and autoflow, both image formats normally provide square outlines. It is possible to develop images in either format that have translucent (or clear) backgrounds to eliminate the square outline. With a translucent image background, the background color of the main window display can show through the translucent areas of the image. You see this technique used particularly with cursive signatures at the bottom of HTML documents, or to make a circular university seal appear circular, instead of circular in a square image. Here's the URL of a document that explains how to do translucent backgrounds:

```
http://melmac.corp.harris.com/transparent_images.html
```

In browsers running on color graphics systems, your images might be sharing the color map with other applications. Mosaic, in particular, only allocates a colormap table with 50 entries. If an inline image has more colors than can be displayed, it still is displayed with the available colors, but it won't look as good as the full-color image. With these colormap issues in mind, you should probably reduce the number of colors used in your GIF files to under 50. If you are going to include multiple GIF images in a single HTML document, you should use the *ppmquantall* utility from the pbmplus package to force all of the images to share a common colormap.* These colormap issues also influence whether you enrich your HTML documents with photographs

* The pbmplus package and its utilities are described in the books *UNIX Power Tools* (O'Reilly & Associates/Bantam Books) and *Encyclopedia of Graphics File Formats* (O'Reilly & Associates, Inc.).

(which require many colors) or with clip-art, which has a more well-defined and limited color palette.

Also think about performance. It takes a lot longer for a Web browser to retrieve an HTML document that has inline images than to retrieve one that does not (unless the user specifies delayed downloading). The larger the inline image, the longer it takes. In fact, the time is proportional to the square of the dimension (a four-inch-square image takes almost twice as long as a three-inch-square one). One thing working in your favor is that many graphically-based Web browsers cache inline images. So if the inline image has been used in previous HTML documents in the user's navigation chain, that image might still be cached. Caching also happens when the same inline image appears multiple times in the same HTML document. It is retrieved only once.

One trick that's used to solve the performance problem of big images is to use "thumbnails". A **thumbnail** is a small version of a figure displayed inline, which is a link to the full-sized image displayed externally. With image manipulation tools such as the pbmplus utilities, it is easy to create a smaller version of an image. The HTML to do this looks like:

```
<A HREF="file-full.gif"><IMG SRC="file-thumb.gif"></A>
```

This HTML uses the syntax for external images, which we'll describe in the next section.

Don't let these issues discourage you from enriching your HTML document with graphics. Use of graphics can add meaning and clarity to the information in your HTML document and can increase the overall usability of the information. You will also see in Section 19.4 that specific areas in inline graphics can become link anchors (clicking on different parts of an image can trigger different links).

19.1.9.2 External images

External images are images that are not displayed inline as part of your document, but in a separate window, by an external viewer program. This is the technique to use if you have TIFF, JPEG, RGB, or HDF images and don't want to convert them to GIF. This technique is also useful for displaying very large GIF images. Instead of using the <IMG> HTML tag described earlier, you simply include the URL of your external image file as part of the hyperlink. For example, to display a JPEG image in an external window, you can include the following in your HTML document:

```
<A HREF="filename.jpg">anchor-text</A>
```

The *filename*.jpeg in this example can be replaced by any URL; it doesn't have to be a local file. See Section 18.3.1 to better understand the handshake between Web server and browser that makes this work. Suffice it to say for now that the user needs an external viewer installed that knows how to deal with the incoming external image file and the Web browser needs to know how to recognize the type of data that's arriving and how to start the appropriate viewer.

Using external images with your HTML document further limits the usability of your HTML document. Just as there are users who can't display inline graphics, there are more who have not configured their environment to support the display of external images.

19.1.9.3 Inline images with character-based Web browsers

Inline images can't be displayed on character-based terminals. Character-based Web browsers indicate that an image is in the autoflow stream by displaying:

```
[IMAGE]
```

However, you can override this default and make your inline images more meaningful in character-based Web browsers by using the ALT option:

```
<IMG SRC="filename.GIF" ALT="image-description">
```

The ALT option makes character-based Web browsers display the image-description instead of [IMAGE]. You can also make the browser ignore an image using a null ALT option:

```
<IMG SRC="filename.GIF" ALT="">
```

When a character-based Web browser sees the null ALT option, it ignores the image-insertion tag. The HTML authoring convention is that if you are going to insert inline images in your document, you should at least describe the image with a word or phrase for users with character-based Web browsers.

19.1.10 Server-Side Includes

The NCSA Web server can include information from files and environment variables into an HTML file just before the file is sent to the user. It also can automatically include the last-modified date, current date, document size, and a few other useful pieces of information. This capability is collectively called **server-side includes**.

Let's look at a quick example. We modify the bottom of our sample HTML template to look like:

```
Creation Date: <EM><!--#echo var="LAST_MODIFIED"--></EM>
<P>
<!--#include file="owner.txt"-->
</BODY>
</HTML>
```

We also create a file called *owner.txt*, which contains a single line:

```
<ADDRESS><A HREF="rjones.html">DRJ</A></ADDRESS>
```

Now all our HTML documents that are based on the sample HTML template will automatically have their last modification date (from the filesystem) and the owner of the file included by the Web server whenever a user requests that file. This guarantees that

the document's modification date is correct—no need to enter it manually. It also gives us the ability to change the owner of every HTML document with a simple edit to *owner.txt*.

Depending on how your Web server is set up, you may have to rename the HTML files that you want processed for server-side includes (usually to use the suffix *.shtml*). Ask your Web server administrator or read Section 18.3.2. It's also possible that the server-side includes feature might not be set up. Or it might have been disabled for security reasons (see Section 21.2.5).

You might have noticed that the ASCII text file *owner.txt* actually contains HTML. *owner.txt* is not called *owner.html*, because it doesn't contain fully structured, legal HTML and because it isn't intended for standalone viewing. (You may want to tell the Web server to ignore included files like *owner.txt* when it does automatic directory indexing, since included files are not intended to be viewed alone. See Section 18.3.3.)

Generically, the format of the include statement is usually:

```
<!--#command tag="value"-->
```

The two commands demonstrated in the HTML above were `#echo` and `#include`. Two others are `#fsize` and `#flastmod`, which include a file's size and last-modified date. The `file=` tag of the `#include` command is used to reference a document in the same physical directory as the HTML document (or a directory relative to that directory). The `var=` tag of the `#echo` command reads and displays the value of an environment variable, which can include any of those defined by the CGI (described in Section 20.1.1), plus the six listed below.

Another `#include` tag (not shown) is `virtual=`, which specifies a file relative to the document root. You would use the `virtual=` tag instead of the `file=` tag if you wanted to have standard boilerplate such as a copyright notice used with every document on your Web server, independent of the document tree structure.

All the tags valid with `#include` are also valid with `#fsize` and `#flastmod`.

The six environment variables (not counting all the CGI variables) you can use with `#echo` are the last modification date (`LAST_MODIFIED`), and:

`DOCUMENT_NAME`
: The current filename

`DOCUMENT_URI`
: The virtual path to this document

`DATE_LOCAL`
: The current date and local time

`DATE_GMT`
: Same as `DATE_LOCAL` but in Greenwich mean time

`QUERY_STRING_UNESCAPED`
: The search query the client sent, if any

There are two more commands not covered here: `#exec`, which runs a script and includes its output, and `#config`, which controls the format of the size and date commands. For complete information, see the URL:

```
http://hoohoo.ncsa.uiuc.edu/docs/tutorials/includes.html
```

19.1.11 For More Information About HTML

How to write forms in HTML (and handle the input that comes from them) is described in Section 20.2.

Appendix A, *gopherd Options*, describes all the tags and options we haven't discussed in this chapter, but doesn't provide any examples.

Not surprisingly, the definitive guide to HTML, *A Beginner's Guide to HTML,* is available on the Web. Point your Web browser at URL:

```
http://www.ncsa.uiuc.edu/General/Internet/WWW/HTMLPrimer.html
```

You should take a look at this document, because it describes all the other HTML tags that are available. It also describes some additional options to the HTML tags that we have outlined here.

Remember also that many Web browsers give you the ability to view the HTML source of the document being displayed, or even to store it locally on your own system. If you see a neat trick that someone else has done with their HTML, just look at the HTML source to see how they did it. In graphical Web browsers, you can also cut and paste from the HTML of existing documents to avoid retyping the tags.

19.2 Information Design Issues

Now that you understand the basics of HTML, you should look at some other issues involved in structuring HTML. This includes how you chop up big documents, how many links you include, how you make those links, and overall style conventions.

19.2.1 Document Modularity

For every document you intend to place on your Web server, you need to think through an assortment of document modularity issues. Should you take documents as they are and convert them directly to HTML? Well, that depends on the length of your documents—three pages (sure), nine pages (maybe), eighteen pages (probably not). Long documents can be slow to download, especially if they include inline graphics.

Do you envision a document broken into a logical hierarchy? Five levels of hierarchy inside your original document does not necessarily have to be broken into five levels of modules with hyperlinks between them. Keep in mind that every time you ask the user to activate a hyperlink, a certain percentage of them will not do it. It is another thing to click on, another couple of seconds to wait (or longer depending on the

bandwidth between your Web server and their Web browser), and there is always the slight possibility that the connection won't go through.

On the other hand, if your document is logically structured into major chapters, it is certainly reasonable to break it into multiple HTML documents (one per chapter or one per major section). If you have sidebars, they too might be broken out from the main flow of the chapters.

You should ask yourself if each HTML document can stand alone or if it is a dangling document that depends on another document for context. Remember that any document on your server might be the first one from your server to be seen. As soon as you place an HTML document on the Web, its URL is available for others to embed in their hyperlinks. On the other hand, people who reference your documents in their HTML have some responsibility for not creating hyperlinks to any dangling documents that might remain.

Many people use a compromise between the single document and multiple document approaches. They have a single HTML document, but provide a table of contents at the start of the document that hyperlinks to other anchors further into the same document. This gives you the best of both worlds. Users can quickly hyperlink to sections if they know what they are looking for or can casually browse the document as a whole. This also establishes alternative entry points into your HTML document, which could be valuable to others who want to reference specific information in your document, but do not want their users to have to wade through the full document.

19.2.2 Degrees of Hyperization

How much time and effort do you direct towards tagging and hyperlinking HTML documents?

On the one hand, you don't have to spend any time at all. You can place ASCII text documents directly on your HTTP directory structure and Web browsers can directly display them. But you lose many of the benefits of the World Wide Web—no autowrapping and autoflowing, no inline graphics, and no hyperlinking to other relevant documents. If you have 2000 ASCII text documents that are preformatted with lots of tables, though, this might not be a bad way to start. At least users can get to your information. It also helps to learn *sed* and/or *awk* to automate the tagging of large numbers of documents.

At the other end of the scale, there is full HTML authoring. Each document is painstakingly crafted with hyperlinks, content is embellished with diagrams and pictures, audio is added where it makes sense, key points of emphasis are highlighted. The reader can more easily grasp the information content and meaning when this is done with care. But it takes time and effort. This level of hyperization also requires maintenance, because links tend to go out of date as Web servers are reorganized and Webmasters come and go. Full authoring is probably worthwhile if you are doing only a single paper or a couple of Web pages. You might also want to take this approach with the

pages that get the most attention in your Web server—your home page, your signature page, and master index pages.

There is also a compromise between these two extremes. The compromise involves the use of the <PRE> tagging pair. Each document to be converted can be divided into two components—the component that is easily converted to HTML and the component that might be too detailed or complicated. The second component can be encapsulated within the <PRE> tagging pair and left alone. The worst scenario in this case is that the entire ASCII text document is encapsulated. This might seem like a waste of time, but the advantage with this approach is that your document is no longer an ASCII text document: it is an HTML document. So the preformatted information in the document can contain hyperlinks.

Documents that never change, such as papers submitted to a conference, have a high payback on the hyperization effort because, once completed, they are never modified and never need to be reconverted. You simply place them online in your Web server and others will read them, enjoy them, and create their own links to them.

19.2.3 Automatic Versus Manual Hyperlinking

Given the primitive state of authoring tools for the creation of HTML documents, most hyperlinks that you embed will be done manually (using a text editor). For frequently accessed documents and major entrypoints into your Web server, that works fine. However, manual hyperlinking becomes tedious when you are dealing with hundreds or thousands of HTML documents.

There are filters and conversion tools for changing other formats to HTML, but there are very few tools that automatically generate hyperlinks in HTML documents. This is primarily because the opportunity to do so tends to be specific to the content found on each Web server. However, if you have a very well-defined information environment you may be able to develop scripts that look for specific instances of phrases—part numbers, people's names, document names, acronyms, etc.—that you might be able to automatically identify, determine the appropriate URL and embed the hyperlink in the HTML document. This technique is also useful if you have preformatted information where each field has well-defined semantics.

HTML conversion tools and filters are discussed further in Section 19.3.5. Often there are cross-references embedded in many of the proprietary document formats. FrameMaker documents and Microsoft Word documents are good examples of this. Usually the HTML conversion filters are able to convert cross-references to hyperlinks when generating HTML.

Embedding absolute URLs (to information resources found outside of your information set) is even more certainly a manual process.

19.2.4 Style Guidelines

You should develop some HTML style guidelines for your site. Consistent HTML style increases the usability of your server—which directly affects how long people spend using your Web server and how often they come back.

Some of the issues are the same as in any authoring environment, but there is a slight twist for the Web. The rules about mixing and matching typefaces go out the window because control of the typeface is not in your hands—it is a Web-browser attribute. However, you still need to consistently use bolding, italics, and other emphasis techniques and to guard against their overuse.

Fortunately, you don't have to start from scratch. People have gone down this path before and have placed their style guidelines on the Web. One of the best is *Style Guide for Online Hypertext* by Tim Berners-Lee of CERN:

```
http://info.cern.ch/hypertext/WWW/Provider/Style/Overview.html
```

Let's go back to our sample HTML document and look at it from a style point of view. Having a standard template will help keep your HTML consistent. We like to have the <TITLE> document title and the <H1> document title match and reinforce one another. The <TITLE> document title is what is placed on a user's hotlist or in their list of bookmarks, while the <H1> document title is the first thing a user sees when the HTML document is displayed.

As we add content, we use <H2> tags to identify major section headings. We also like to frame our documents with a horizontal rule (<HR>) at the top and bottom of the page. This visually separates the document content from title information on the top, and from version control and author signature on the bottom.

Your overriding concern should not be what your style guidelines are, but that you have some and use them! If you are consistent, it is not too difficult to make global changes.

19.2.5 An Example of Information Design

To illustrate these HTML concepts we will now show a more complicated HTML file: the HTML we set up for presenting an Action Plan developed by the Species Survival Commission.

We decided to focus the most important components of the document on the home page. We did not try for a one-to-one mapping between the cover of the paper document and the top-level HTML document. In addition to capturing the document title, cover photo, and author/editor information, we also opted to include the copyright information (found on the inside cover) and the main section titles from the table of contents.

To capture the photo, the document was taken to the local desktop publishing center and the crocodile picture was scanned in as it appears (6″ × 4″) and returned in GIF

format at 200 dpi. *xv* was used to try several scaling factors. We settled on 20 percent of full scale as the best size for the home page. By scaling the crocodile photo to that size and saving it in GIF format, we were able to include it as an inline graphic.

The home page HTML contents look like this:

```
<HTML>
<HEAD>
<TITLE>Crocodiles: An Action Plan for their Conservation</TITLE>
</HEAD>
<BODY>
<H1>Crocodiles</H1>
<H2>An Action Plan for their Conservation</H2>
<IMG SRC="crocodile.gif" ALT="Picture of a Crocodile">
<H2>Table Of Contents</H2>
<DL>
<DT><A HREF="foreword.html">Foreword & Acknowledgements</A></DT>
<DT><A HREF="executive.html">Executive Summary</A></DT>
<DT><A HREF="objectives.html">Objectives and Organization</A></DT>
<DT><A HREF="intro.html">Introduction and Conservation Priorities</A></DT>
<DT><A HREF="countries.html">Country Accounts</A></DT>
</DL>
<P>
<HR>
<P>
<EM>&#169 1992 International Union for Conservation
of Nature and Natural  Resources<EM>
<P>
Compiled by <A HREF="jthor.html">John Thorbjarnarson</A><BR>
Edited by <A HREF="hmessel.html">Harry Messel</A>,
<A HREF="wking.html">Wayne King</A>, and
<A HREF="jross.html">James Perran Ross</A>
</BODY>
</HTML>
```

We started with the HTML document template shown earlier. We decided not to use two horizontal rules, as it just seemed too complicated-looking for the home page. Relative URLs were used to create the hyperlinks from the table of contents to the subordinate documents. We used the ALT option with the <IMG> image insertion tag for the benefit of line-mode Web browsers. If the Web browser can't display the inline image, it inserts the phrase "Picture of a Crocodile" instead.

Figure 19-4 shows how this HTML document looks when displayed with Mosaic. To reinforce the differences in look and feel between Web browsers, we've also shown in Figure 19-5 how this same document looks when displayed in a terminal window with Lynx.

You probably also spotted a new HTML technique used in this example. We used a special insertion code (&#169) to produce the copyright symbol, shown at the bottom

of Figure 19-4. For more information and examples of all the ISO 8859 insertion codes, point your Web browser at:

```
http://www.uni-passau.de/~ramsch/iso8859-1.html
```

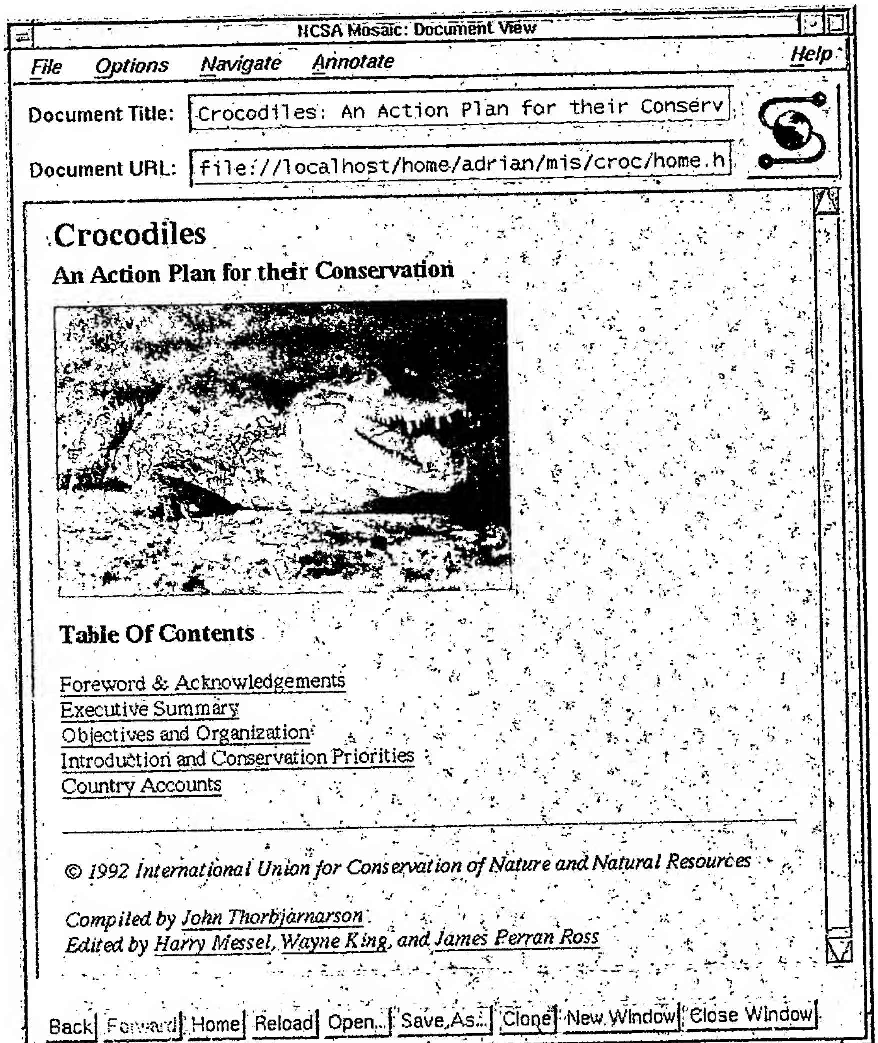


Figure 19-4. Crocodile home page, displayed with Mosaic

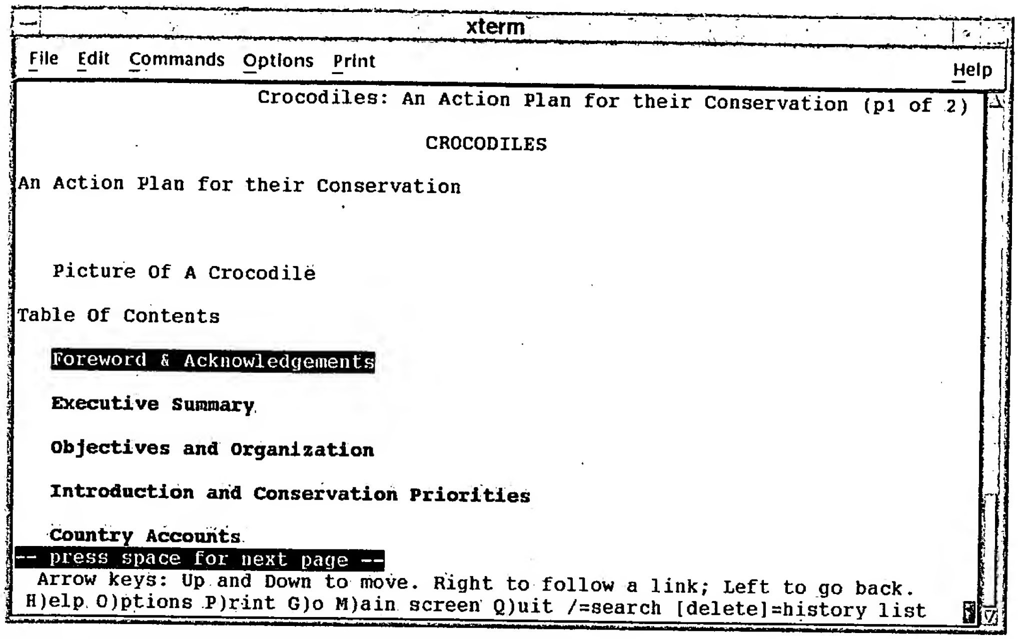


Figure 19-5. Crocodile home page, displayed with Lynx

19.3 HTML Authoring Tools

The subject of HTML authoring tools is the most undeveloped area of the World Wide Web. HTML authoring tools differ from HTML conversion tools (listed in the next section) in that the documents are built from the ground up and have some notion of global hyperlinking with URLs. There are four ways to author HTML documents:

- You can directly embed HTML tags with your favorite text editor or perhaps you can get some assistance from the HTML mode available for Emacs.
- You can use one of the free HTML editors available for the X Window System. One called *htmltext* supports WYSIWYG HTML editing. For more information, point your Web browser at:

  `http://web.cs.city.ac.uk/homes/njw/htmltext/htmltext.html`

  Don't forget that TkWWW (mentioned in Section 17.3.1.8) is an HTML editor. However, because it's also a Web browser, you can try out links immediately after creating them.
- You can use a traditional commercial publishing package such as FrameMaker and then convert your documents to HTML format.
- You can use one of the emerging commercial HTML authoring tools for the Web such as HoTMetaL from SoftQuad or CyberLeaf from Interleaf.

The NeXT-based Web browser developed at CERN has HTML authoring features. Unfortunately, you have to have the orphaned NeXT workstation to use it.

Whatever authoring technique you choose, be sure to preview your HTML with an assortment of Web browsers. A bug in your HTML may appear in some browsers, but not others. When the SGML-compatible version of HTML becomes standard, you will be able to validate your HTML with a special tool to make sure it is syntactically correct without using multiple browsers.

19.3.1 Emacs HTML Mode

Emacs HTML mode *(html-mode.el)* runs under GNU Emacs v18 and v19, Epoch, and Lucid Emacs. This extension allows you to use keyboard commands to prompt and format many of the major HTML constructs: title, headings, paragraphs, different types of lists, anchors, and IMG tags. It supports Lucid Emacs menubar and font-lock capabilities.

To get the Emacs HTML-mode extension, point your Web browser at:*

```
ftp://ftp.ncsa.uiuc.edu/Web/html/elisp/html-mode.el
```

19.3.2 FrameMaker

If you already have FrameMaker, it might be easier to author with an HTML document style template, save the document in Maker Interchange Format (MIF), and then convert to HTML. The following URLs identify various filters and templates that are available for FrameMaker authoring:

```
http://info.cern.ch/hypertext/WWW/Frame/fminit2.0/www_and_frame.html

http://www1.cern.ch/WebMaker/WEBMAKER.html

ftp://bang.nta.no/pub/

ftp://ftp.alumni.caltech.edu/pub/mcbeath/web/miftran/
```

19.3.3 HoTMetaL

SoftQuad HoTMetaL is a WYSIWYG HTML editor that is available for Sun/Motif and Microsoft Windows. SoftQuad has established a collaborative relationship with NCSA, and the two organizations are integrating HoTMetaL and Mosaic for better interoperation. Because it is context-sensitive, HoTMetaL guides authors in creating new HTML documents and cleaning up old ones.

* Note: all the URLs in this book are included in a file in the book's code archive. If you get this file, it can save you from retyping the URLs and also lets us update the ones that change.

HoTMetaL comes in two flavors, an unsupported free version and an enhanced commercial version. The free version can be retrieved by pointing your Web browser at:

```
ftp://ftp.ncsa.uiuc.edu/Web/html/hotmetal
```

SoftQuad HoTMetaL Pro is a fully-supported commercial version of HoTMetaL that handles more complex document structures, such as tables and forms. It is currently available for UNIX; support for additional platforms is forthcoming. For more information, send email to *hotmetal@sq.com*.

Simultaneously with the announcement of SoftQuad HoTMetaL, Avalanche Development announced companion conversion tools that convert WordPerfect and Microsoft Word documents to HTML.

19.3.4 CyberLeaf

Interleaf's CyberLeaf product is billed as a commercial-grade tool for HTML document creation, assembly, and conversion. Interactive resolution of URLs provides live integration of the CyberLeaf authoring environment with the Web. CyberLeaf features point-and-click hyperlinking within and between documents.

CyberLeaf also provides interactive or batch-mode conversion of existing Interleaf documents to HTML. It supports an extensive collection of bidirectional and compound filters for Microsoft Word and WordPerfect documents. For more information, send email to *pwerner@ileaf.com*.

19.3.5 HTML Conversion Tools and Filters

HTML conversion tools and filters are much more numerous than native HTML authoring packages. To learn more about some of the more popular tools, point your Web browser at the URLs listed below to either retrieve the tool (and look at the *README* file) or learn more about the tool via its online description.

In many cases, there are multiple versions of some conversion tools. An original version was created and then others used that as a starting point to add different features. You can also find out the author of each tool and correspond with them on bug fixes and suggestions for additional functionality.

ASCII Text to HTML:

```
http://www.seas.upenn.edu/~mengwong/txt2html.html
```

BibTeX to HTML:

```
ftp://gaia.cs.umass.edu/pub/hgschulz/windex-1.2.tar.Z
http://www.research.att.com/biblio.html
```

Interleaf to HTML:

```
http://info.cern.ch/hypertext/WWW/Tools/il2html.html
http://info.cern.ch/hypertext/WWW/Tools/interleaf.html
```

LaTex to HTML:

```
http://cbl.leeds.ac.uk/nikos/tex2html/doc/latex2html/latex2html.h tml
ftp://skye.aiai.ed.ac.uk/pub/tex2rtf/
```

Mail to HTML:

```
ftp://ftp.uci.edu/pub/dtd2html/
```

Look in the directory for the latest *perlWWW tar* file.

Mosaic Hotlist to HTML:

```
ftp://ftp.uci.edu/pub/dtd2html/
```

Look in the directory for the latest *perlWWW tar* file.

PostScript to HTML:

```
http://stasi.bradley.edu/ftp/pub/ps2html/ps2html-v2.html
```

Texinfo to HTML:

```
http://asis01.cern.ch/infohtml/texi2html.html
```

Troff to HTML:

```
http://cui_www.unige.ch/ftp/PUBLIC/oscar/scripts/ms2html
```

To check the latest information on conversion tools and filters, point your Web browser at:

```
http://info.cern.ch/hypertext/WWW/Tools/Filters.html
```

19.4 Clickable Image Maps

Let's say you want people to be able to get their local weather from your Web server. What would be better than to show them a map of the country (inline), and allow them to click on any location to get that location's weather? That's what a clickable image map does. It has many other uses, for example:

- To help users better understand a picture or diagram by allowing them to click on a component represented in the graphic to see an explanation
- To let users move down to finer levels of detail on a map of a campus or building

Clickable image maps only work with graphically-oriented Web browsers. You should always try to provide alternate text-based methods of accessing the same information.

There are several well-defined steps you need to take when creating a clickable image map. The steps are:

1. Get an image and figure out where to put the **hotspots** (the areas that trigger links)
2. Develop an image-map file that describes where each hotspot is located and specifies the URL associated with each hotspot
3. Tell your Web server how to reach the image-map file
4. Reference the image map in an HTML document
5. Test the clickable image map

Let's take a look at each of these steps in detail.

19.4.1 Planning the Clickable Image Map

First select the inline image you will use. As with any inline image, the graphic needs to be in GIF format for maximum portability. Most graphics service bureaus can scan photographs to create GIF files, or you can draw the graphic with a paint program.

Then rough out mentally where you will place the hotspots. Let's take a look at an example. Figure 19-6 shows a photo of my office. I'm going to use this as the basis for creating a virtual office. With the blurring between the physical world we live in and the virtual online world found on the Web, I should be able to have some fun with this photo.

Arbitrary rectangles, polygons, and circles can be used to identify the hotspots. Figure 19-6 shows how the hotspots in this image are conceptualized. I can directly hyperlink from some of the physical things found in the photo to their virtual equivalents found out on the Web. For example, I placed a copy of *Wired* magazine on my table. I established a hyperlink from the magazine in the photo to *Wired* magazine's Web server. Where virtual equivalents don't exist, I can create them.

Which shapes should be used depends on what your graphic represents:

- A map of an office floor plan might represent each office as a rectangle.
- A map of a class photo might represent each person's head as a circle.
- A map of an airplane might represent the jagged outlines of the wing or a tail section as a series of polygons.

19.4.2 Mapping the Hotspots into a Map File

The second step is to develop an image-map file. You need to create a separate image-map file for each clickable image map on your Web server.

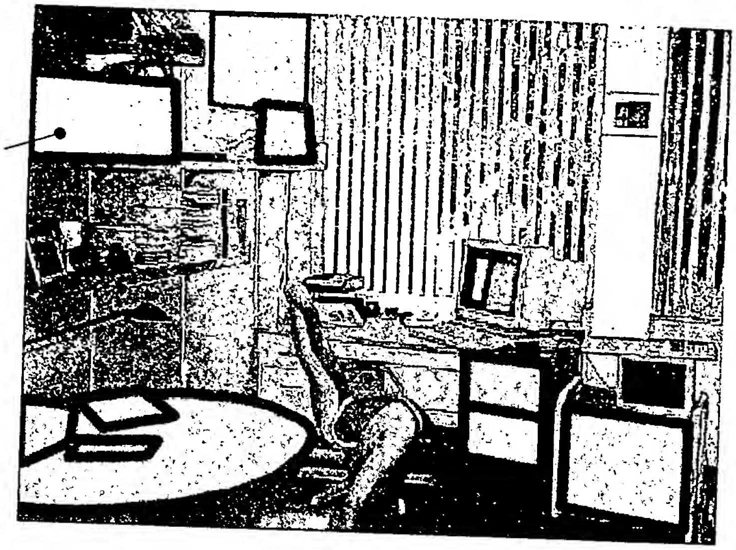


Figure 19-6. A digitized photo of my office

Each line in an image-map file defines a hotspot, by defining an area within the graphic and the corresponding URL to be returned if someone clicks on that area. You can include comment lines by beginning the line with the pound (#) sign.

Hotspots are formatted as:

```
shape URL coordinate-1,coordinate-2 ... coordinate-n
```

where shape is one of the following:

circle A circle (with two coordinate pairs: center and any edgepoint)

poly A polygon (with at most 100 vertices; each coordinate is a vertex)

rect A rectangle (with two coordinate pairs: upper-left and lower-right)

Coordinates are x,y pairs counting in pixels from the upper lefthand corner of the image.

The URL can be either an actual URL pointing to any resource on the World Wide Web or a URL on your server, but without the *http://hostname*.

Let's look at the image map that I put in the file *office.map*:

```
default /map/none.html
poly /map/upper-right-drawer.html 423,319 423,352, 499,359, 499,326
poly /map/lower-right-drawer.html 423,352 422,416 498,425 499,359
poly /map/home-page.html 425,220 419,226 442,271 447,222
poly /map/workstation.html 519,364 517,460 633,478 638,376
rect /map/bookshelf.html 0,60 139,143
```

```
poly http://www.wired.com/ 51,357 77,393 144,385 115,365
poly http://nearnet.gnn.com/gnn/gnn.html 55,413 46,429 120,425 126,410
poly /map/books-on-table.html 0,375 0,436 45,413 46,380
poly /map/t.html 0,395 0,467 155,458 230,436 273,401 225,371 153,364 67,372
rect /map/stereo-speaker.html 221,90 268,140
poly /map/wall-picture.html 167,0 171,89 226,96 263,0
```

The server takes the coordinates from a user's mouse click and steps through the map file to determine if the click is within any hotspots. As soon as the first match is found, its corresponding URL is used to redirect a document to the user's Web browser. If no matches are found, a default URL (that you specify in the image-map file) is returned. There can be only one default per map file.

It is normal to mix and match circles, multiple polygons, and assorted rectangles in the same map file. Although you should try to minimize overlapping hotspots in the image map, if there are any, the first match is the one used. Just make sure they are ordered the way you want them.

How do you determine the x,y pixel coordinates to go into each entry? One way is to use *xv* to view the image and then just click on the graphic to get the coordinates needed according to the shape you want to use for each hotspot.

Note that all the URLs given in the map file point to HTML files, not graphic files, even though some of them actually display pictures. I could have had the URL for the photo on the wall point directly to a GIF file of that photo, but I didn't. That would have made the assumption that the user has an external viewer capable of handling GIF files (because only graphics invoked from within HTML documents are shown inline). Instead, I have each hotspot point to a *.html* file. Then each *.html* file handles the graphic as an inline image.

For default URLs (for when someone clicks on a spot that isn't hot), the virtual office returns an error message along with a smaller version of the same photo of the office with the hotspots marked. Actually, it's a lot like Figure 19-6. Don't return a complicated graphic that only says "sorry; try again." On a slow link, the error message/graphic can take quite a while to retrieve, and it can be frustrating when the user sees that the graphic it took 30 seconds to retrieve wasn't what he wanted. Or use a simple text error message if the default is an error message.

19.4.3 Connecting Your Map File to a URL

The next step ties your map file to the map name that you will use in your HTML. The *imagemap.conf* configuration file contains a list of all your server's map-file locations and their corresponding names. It is usually found in the */conf* directory in your Web server root directory. In *imagemap.conf*, lines beginning with a pound sign are comments. Every other nonblank line is an entry that consists of:

```
name: path
```

Where *name* is the map name and *path* is the absolute path to an image-map file, or a relative path from the Web server root directory. In the example we have used so far, you would include the following map entry in the *imagemap.conf* file:

```
office: /map/office.map
```

The *imagemap.conf* file is not loaded when you start your *httpd* server. It gets loaded every time someone clicks on a clickable image map.

You can place map files anywhere you want to on your server. You may want to keep the map files for one server all logically together in the same */map* directory or you may want to encourage individuals to keep their image maps together with the images and any supporting HTML documents in the same directory structure.

19.4.4 Referencing Your Clickable Image Map in HTML

We discussed earlier how to include inline graphics in your HTML document using the image insertion HTML tag:

```
<IMG SRC="URL">
```

The final step in creating an image map is to tell the Web browser accessing the HTML document that the inline image it will display is a clickable image map. This is done by adding the optional ISMAP qualifier to the <IMG> tag:

```
<IMG SRC="URL" ISMAP>
```

In practice, what you do is create a link where this image is the link trigger and the URL invokes the imagemap script and passes it the map name:

```
<A HREF="/cgi-bin/imagemap/map-name"><IMG SRC="URL" ISMAP></A>
```

We'll explain how this actually works in a moment. But all you really need to know is to use exactly this HTML, substituting your map name (from before the colon in *imagemap.conf*) and your URL. In our example, we would reference the virtual office image map with the following HTML:

```
<A HREF="/cgi-bin/imagemap/office">
<IMG SRC="/map/office.gif" ISMAP></A>
```

19.4.5 Testing the Image Map

Now, if the magic works, the image map should be working. First we open the URL of the document containing the HTML just shown. An inline image should appear in the browser (see the left side of Figure 19-7).

Clicking on the bookshelf on the left wall of the office brings up a bibliography of the books on the shelf, as shown on the right in Figure 19-7.

Clicking on one of the books retrieves information about the book from the publisher's Web servers.

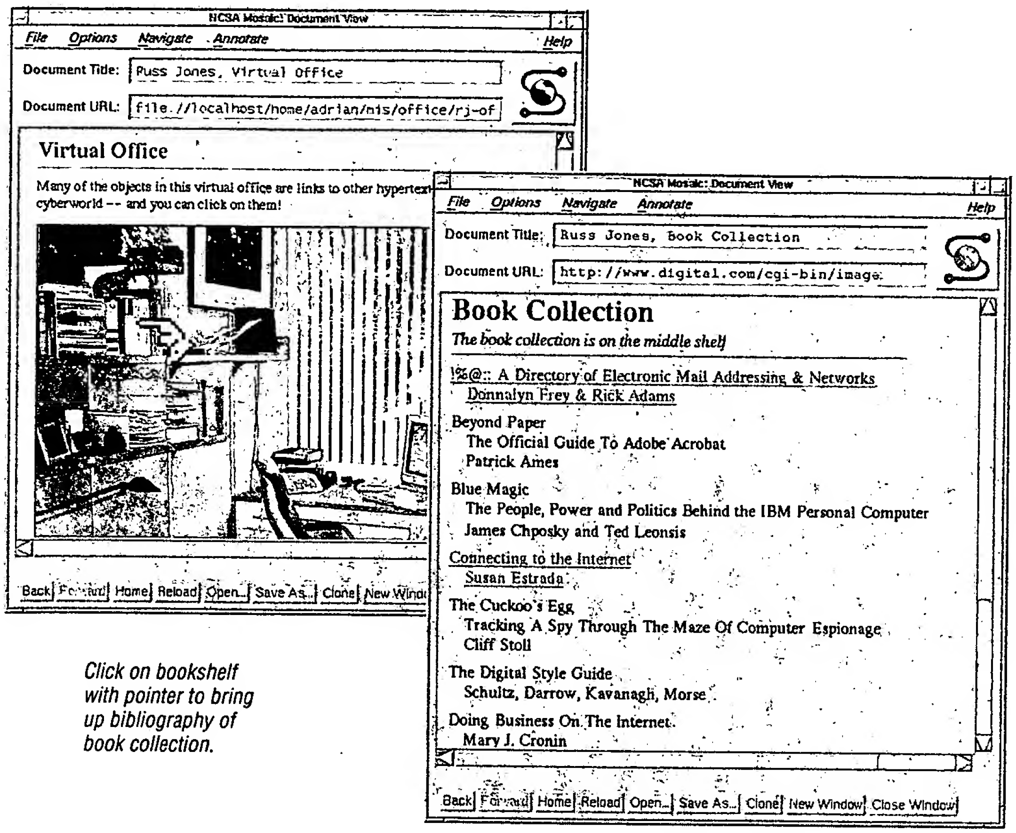


Figure 19-7. The virtual office clickable image map in Mosaic

19.4.6 How It Really Works

imagemap is a precompiled gateway that is included in the *cgi-bin* directory with the Web server distribution.*

Here's how an image map actually works. When the user clicks on the image, it triggers the link, which makes the Web server invoke the *imagemap* program, passing it the map name. The *imagemap* program opens *imagemap.conf* and uses the map name to get the right map file. Then it opens the map file and searches for the first hotspot that matches the button click. When it finds one, it returns the matching URL.

* If you have changed the default directory where the *imagemap.conf* configuration file is kept to anything other than */usr/local/etc/httpd/conf*, then you need to change the *imagemap.c* source and recompile it to reflect the actual location of the *imagemap.conf* file.

One revelation you get by understanding how it works is that two files are opened and read for every click on an image map. There is also some processing to determine which hotspot matches the click. So image maps are not free, and you should take care to keep *imagemap.conf* and your map files as short as possible.

In this Chapter:
- *Gateways*
- *Forms Processing*
- *WAIS Access from the Web*

20

Web: Gateways and Forms

Web browsers can directly access several types of Internet information services, but not every type. For example, Archie and Finger are not supported. Furthermore, not all local information sources fit the Web authoring mold in which you create static files and place them in directories. Perhaps the information is dynamically generated, or comes from an ORACLE database.

Gateways solve these problems by providing an extension mechanism for the Web server. A gateway takes an information source that doesn't fit the mold and makes it look to the browser like a file on the Web server. In practice, a gateway is just a script or program invoked by your Web server, that can accept user input through the Web server and can output HTML, a URL, or some other data back to the user through the Web server.

A form provides a nice familiar way to collect data from the user. The Web server passes the form input to a script or program on your system, which can do whatever you want with the input. Form scripts are closely related to gateways, because the scripts or programs in both cases pass data to and from the Web server in the same way.

The input provided by a form can be used by a gateway to get information, or the input can just be filed or emailed to someone. For example, O'Reilly & Associates uses a form to allow people to subscribe to its *GNN* online magazine on the Web.

While some Web browsers provide direct access to WAIS databases, most don't. So you may want to provide a gateway to your WAIS server. This chapter describes the four ways of accessing WAIS servers and explains how to pick one.

20.1 Gateways

You can use gateways running on the Internet, install gateways that have been developed by others, or develop your own. You don't always have to set up your own. But if you want better performance or you want to control the formatting of the information that is displayed to a user, you need to run the gateway and control the HTML it automatically generates.

20.1.1 The Common Gateway Interface

The *Common Gateway Interface* (CGI) is the mechanism for communicating between your gateway and your Web server.

Figure 20-1 illustrates the basic information flow through the CGI, the Web server, and the client. When the user enters text on a form or in response to <ISINDEX> query and hits the return key, the Web browser sends keystrokes captured from the user to the *httpd* server. The *httpd* server accepts the input, starts up the gateway and hands the input to the gateway via the CGI. The user's keystrokes are passed to the gateway either via environment variables (called the GET method) or using standard input (called the POST method.) The gateway then parses the input and processes it. It may generate HTML output, which is returned to the *httpd* server to pass to the client, or it may save data in a file or database or send email to someone.

The gateway can be a script or programs, written in C/C++, Perl, tcl, the C Shell, or the Bourne Shell. Each language has its own strengths as a gateway language.

CGI gateways that generate HTML output are required to preface the HTML output to *stdout* with the following line:

```
Content-type: text/html
```

This line must be followed by a blank line before the first <HTML> tag is sent.

The gateway does not have to generate HTML. It could return the URL of another file, indicating to the browser that it should get that file. This is called *URL redirection*. CGI gateways using URL redirection write the following line to *stdout*:

```
Location: URL
```

This line, too, must be followed by a blank line before the *stdout* data stream terminates.

20.1.1.1 Sample gateways

The NCSA Web server kit is distributed with a set of sample gateways. Some are directly useful, while others should help you understand CGI scripts and help you create custom gateways. The following CGI test scripts are located in *cgi-bin* (the language they are written in is shown in parentheses):

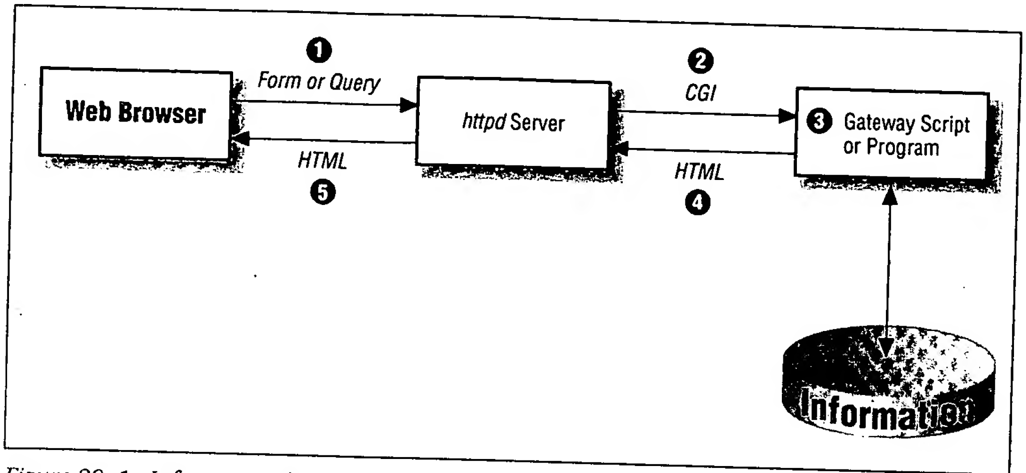


Figure 20-1. Information flow through the Common Gateway Interface

- *test-cgi* (sh)
- *nph-test-cgi* (sh)
- *test-cgi.tcl* (tcl)

There are also a number of simple gateways in *cgi-bin*:

archie (sh)	A very simple gateway to Archie
calendar (sh)	An interface to the UNIX *cal* command
date (sh)	Prints the current date
finger (sh)	A simple gateway to Finger
fortune (sh)	Random fortune teller

In the *cgi-src* directory are source files that show how to implement a custom gateway as a program:

uptime (sh)	Prints the server's load average
jj (C)	A sample HTML form to order a submarine sandwich
phf (C)	An HTML form interface to CSO phone book servers
query (C)	A general-purpose form response script
imagemap (C)	Handles clickable image-map queries
wais.pl (Perl)	One type of WAIS gateway

Section 14.1 and Section 19.4 describe how to use the *wais.pl* script and the *imagemap* gateway, respectively.

20.1.1.2 How scripts work

Let's try the *test-cgi* script. Assuming your Web server is running, and you haven't modified the *ScriptAlias* directive that defines the *cgi-bin* directory, the script is invoked by opening the following URL:

```
http://hostname/cgi-bin/test-cgi
```

When invoked, it should produce HTML output that describes how your *httpd* server is configured and what information is being passed generically to all CGI scripts. The output should look like this in a browser:

```
CGI/1.0 test script report:

argc is 0. argv is .

SERVER_SOFTWARE = NCSA/1.3
SERVER_NAME = hostname
GATEWAY_INTERFACE = CGI/1.1
SERVER_PROTOCOL = HTTP/1.0
SERVER_PORT = 80
REQUEST_METHOD = GET
HTTP_ACCEPT = text/plain, application/x-html, application/html,
text/x-html, text/html, image/*, application/postscript, video/mpeg,
audio/basic, audio/x-aiff, image/gif, image/jpeg, image/tiff,
image/x-portable-anymap, image/x-portable-bitmap, image/x-portable-graymap,
image/x-portable-pixmap, image/x-rgb, image/rgb, image/x-xbitmap,
image/x-xpixmap, image/xwd, image/x-xwd, image/x-xwindowdump, video/mpeg,
application/postscript, application/x-dvi, message/rfc822, application/x-latex,
application/x-tex, application/x-texinfo, application/x-troff,
application/x-troff-man, application/x- troff-me, application/x-troff-ms,
text/richtext, text/tab- separated-values, text/x-setext, */*
PATH_INFO =
PATH_TRANSLATED =
SCRIPT_NAME = /cgi-bin/test-cgi
QUERY_STRING =
REMOTE_HOST = xxx.xxx.xxx
REMOTE_ADDR = xxx.xxx.xxx.xxx
REMOTE_USER =
AUTH_TYPE =
CONTENT_TYPE =
CONTENT_LENGTH =
```

All of the capitalized variable names (like SERVER_NAME) are environment variables that are set by the CGI. These are available to be used by your gateway.

test-cgi is an example of a gateway that requires no input, but it does print out any input you give it. In this case, `argc is 0. argv is .` means we didn't give it any. However, parameters can be passed to a gateway by appending them to the URL using the following scheme:

```
URL?first+second+third
```

For example, to pass your name (*John Q. Public*) to the *test-cgi* gateway, you would open the following URL:

```
http://hostname/cgi-bin/test-cgi?John+Q.+Public
```

The incoming parameter is appended to the base URL after the ? character, with blanks replace by the + character. The parameter is set as an environment variable called QUERY_STRING, but is not passed to *test-cgi* as a command-line option, or in standard input. The HTML output returned from this URL would be exactly as shown previously, except for:

```
argc is 3. argv is John Q. Public.

QUERY_STRING = John+Q.+Public
```

Passing parameters to CGI scripts by appending them to the URL is referred to as the GET method. CGI scripts written to use the GET method must explicitly read the QUERY_STRING variable to get and then parse the query. The GET method is usually used to invoke a simple CGI script with a single parameter.

Alternatively, parameters can be passed to CGI scripts using standard input. This is called the POST method. It is used when there is a large amount of information that would be awkward to append to the end of the URL (as with the GET method) or when the parameters need to be encoded. Most of the advanced work on secure transactions is focused on secure encoding techniques for the POST method. The POST method is generally used with forms processing. See Section 20.2 for more information.

When using the POST method, two parameters, for a user's name and email address, might look like:

```
NAME=John+Q.+Public&EMAIL=jpublic@netcom.com
```

The CGI script would then parse the two parameters as:

```
NAME     John Q. Public
EMAIL    jpublic@netcom.com
```

The & character separates the parameter pairs and the = character separates each parameter name from its value. The + character is used again as a substitute for blanks. The Perl CGI library contains code to parse standard input to load an associative array for processing.

20.1.2 Prompting the User for Input

For most CGI scripts that search or return selected data, you will prompt the user for input to pass to the script. This is accomplished by using the <ISINDEX> HTML tag. When an HTML document contains an <ISINDEX> tag, the browser displays an input box with the phrase:

This is a searchable index. Enter search keywords: <type-input-here>

This does not mean your HTML document is automatically a searchable index. The <ISINDEX> tag just captures user keystrokes and sends those keystrokes to a gateway using the GET method. The gateway performs the actual search.

If the gateway doesn't exist, placing the <ISINDEX> tag in the HTML document will not make it exist, although it will look to the user as though it exists. The user can certainly key in a search string. But when the Return key is hit, no search will occur. To have your Web server perform search functions you must develop or set up a gateway to perform the search.

20.1.2.1 Simple search gateways

The trick to developing your own simple search gateway is to use a dual-purpose CGI script. It is dual-purpose because when invoked with no argument, the script asks for the search string. When invoked with arguments (appended to the URL), the arguments are taken as the search string, and the script executes the search.

For example, the following URL might be used to prompt for a search phrase:

```
http://www.fredonia.edu/cgi-bin/simple-search
```

This URL invokes a script called *simple-search*. The following HTML shows you how to reference the simple search script as part of a hyperlink:

```
Or you can <A HREF="/cgi-bin/simple-search">search</A> to
locate the latest information.
```

The following HTML shows you how to reference the simple search script with a predefined search phrase:

```
Or you can search the database for a list of
<A HREF="/cgi-bin/simple-search?crocodile">crocodiles</A> that are
endangered.
```

Notice that the search phrase is appended to the URL after a question mark. Let's look at the two functions of the simple search script more closely. The logic of the simple search is:

- Was the CGI script invoked with the search string appended to the URL?
- If not, then:
  1. Generate the initial query request using the <ISINDEX> HTML tag.
  2. Write the query request to *stdout*.
- If so, then:
  1. Get the search phrase from the QUERY_STRING environment variable.
  2. Parse the query string to convert the search phrase back to English.

3. Invoke the search.
4. Write the results to *stdout.*

Before we show a script that does this, we need to say a few words about security so you'll be better able to understand the script.

20.1.2.2 Scripts, programs, and security

As mentioned earlier, you can develop gateways using almost any scripting language or programming language.

You need to be aware of security holes that you could introduce in your Web server if you are not careful in writing your CGI script. Anytime you capture user keystrokes, those keystrokes could be malicious. Someone attacking your Web server can embed shell metacharacters in their input that could result in shell syntax errors or the execution of arbitrary commands on the system that runs your gateway. The danger is particularly common with CGI gateways written in the Bourne or C Shell, in Perl, or in any language where an interpreter can execute commands external to your gateway.

This doesn't mean that you shouldn't write a CGI gateway in these languages, just that you should be careful. Fortunately, people have developed callable libraries that make sure keystroke input is clean of malicious metacharacters. The CGI library for Perl is available from:

```
ftp://ftp.ncsa.uiuc.edu/Web/httpd/Unix/ncsa_httpd/cgi/cgi-lib.pl.Z
```

When we write CGI scripts, rather than trapping "problem" metacharacters, we simply use the following regular expression to check for legal keystrokes:

```
[a-zA-Z0-9_-+ \t\/@%]
```

It's hard to tell, but there is a blank space after the plus (+) sign. Blanks are certainly safe! If you are developing CGI scripts in Perl, the code fragment to validate input might look like:

```
if ($variable-name !~ /^[a-zA-Z0-9_-+ \t\/@%]+$/) {
    &report_evil_characters;
    exit;
}
```

variable_name is the name of the variable that holds the typed-in characters. *report_evil_characters* is a subroutine that formats an error message in HTML to be sent to the user. You do not have validate every value—just the ones you are going to pass directly to interpreters outside of your CGI script.

20.1.2.3 Search gateway example

Now we'll show you the search gateway that implements the logic described in Section 20.1.2.1 and accounts for the security concerns described in Section 20.1.2.2. This gateway is called *alias-search* and shows how a CGI script can be used to search the system *aliases* file for email aliases. The following example shows the Perl code used to implement the gateway.

You can easily modify this example to search data files other than */etc/aliases*. If the data you are searching has a predetermined format or syntax, you can refine this example by getting rid of the <PRE> tagging pair and using other HTML tags to format and add emphasis to the search results. This same general search model can also be used for forms processing, which will be discussed in Section 20.2.

```
#!/usr/local/bin/perl

$data_file = "/etc/aliases";

# make sure arguments are passed using the GET method
if ($ENV{'REQUEST_METHOD'} eq 'GET') {
    # Get the query in phrase1+phrase2 format from the QUERY_STRING
    # environment variable.

    $query = $ENV{'QUERY_STRING'};

    # If the query string is null, then there is no search
    # phrase appended to the URL
    if ($query !~ /\w/) {

        # No argument, so prompt the user for the search string
        # using the <ISINDEX> HTML Tag.
        &html_header("Alias Search Query");
        print "<ISINDEX>";
        &html_trailer;
    }
    else {
        # The search string is appended to the URL.  Massage it
        # back into a useable form.

        $query =~ tr/+/ /;
        $query =~ s/%([a-fA-F0-9][a-fA-F0-9])/pack("C", hex($1))/eg;
        # Check for illegal metacharacters

        if ($query !~ /^[a-zA-Z0-9_-+ \t\/@%]+$/) {
            &html_header("Illegal Characters");
            print "The search phrase contains illegal \n";
            print "characters. Please back up and resubmit \n";
            print "the query.\n";
            &html_trailer;
        }
        else {
            # Start the HTML stream back to the Web server.
```

```
            &html_header("Alias Search Results");
            print "<PRE>\n";

            # Open the datafile and search through it.
            # Write out any matches to stdout.

            open(DATAFILE,"$data_file");
            while (<DATAFILE>) {
                if (/$query/i) {print;}
            }
            close(DATAFILE);
            # terminate the HTML stream back to the the Web server.

            print "</PRE>\n";
            &html_trailer;
        }
    }
}

# =====================================================
# This subroutine take a single input parameter and uses
# it as the <TITLE> and the first level header.
# =====================================================
sub html_header {
    $document_title = $_[0];
    print "Content-type: text/html\n\n";
    print "<HTML>\n";
    print "<HEAD>\n";
    print "<TITLE>$document_title</TITLE>\n";
    print "</HEAD>\n";
    print "<BODY>\n";
    print "<H1>$document_title</H1>\n";
    print "<P>\n";
}
# =====================================================
# This subroutine finishes off the HTML stream.
# =====================================================

sub html_trailer{
    print "</BODY>\n";
    print "</HTML>\n";
}
```

The line:

```
if ($ENV{'REQUEST_METHOD'} eq 'GET') {
```

verifies that the CGI script was invoked using the GET method (not the POST method). Well-written CGI scripts handle invocation using either CGI method (GET or POST). Less well-written CGI scripts handle one or the other, but at least verify the method at run-time. Poorly written scripts do not check the CGI method and bomb at run-time.

20.1.3 URLs in HTML Documents Returned by Gateways

HTML documents returned by CGI gateways can contain absolute URLs for all hyperlinks. Relative URLs won't work by themselves, because there is no "current document" from which to fill in the access method and server name.

If you want to use relative URLs, the <BASE> HTML tag explicitly states the base URL of the HTML document containing the relative URLs. Here's the syntax of <BASE>:

```
<BASE HREF="URL">
```

The <BASE> tag specifies the URL of the document itself, to be used when the document is read out of context, such as by a gateway. Relative URLs within the document will be resolved relative to this base URL, as opposed to the URL used to reach the document. There can only be one <BASE> tag per HTML document, and it is included in the <HEAD> section.

The use of the <BASE> tag negates the portability of the HTML, but you still have the simplicity advantage of specifying URLs relative to your document tree. However, the <BASE> tag is not consistently implemented across all Web browsers yet, so you may want to use this with caution. (Mosaic for UNIX supports it. Mosaic for the Macintosh and Windows does not, nor does Lynx. However, it's part of the HTML standard.)

20.1.4 Existing Gateways

People contributing to the World Wide Web have created many of the obvious gateways to common resources found on the Internet. The following gateways are freely available and can be installed on your Web server as needed. For information on the WAIS gateway, see Section 20.3.

Hytelnet Gateway

Hytelnet provides menu-driven access to Internet-accessible *telnet* sites. *Hytelnet* lists libraries, Campus-Wide Information Systems, and Freenets around the world. This gateway provides run-time conversion of the *Hytelnet* database to HTML.

```
http://info.cern.ch/hypertext/WWW/HytelnetGate/Overview.html
```

Author: Earl Fogel *<earl.fogel@usask.ca>* at Computing Services, University of Saskatchewan.

man2html Gateway

man2html takes *man* pages in formatted *nroff* and outputs HTML. The *nroff* output is surrounded with <PRE> tags, with the exception of section heads and overstruck words, which are marked up with the appropriate header level and emphasis.

man2html can be used as either a conversion filter or installed as a gateway. By installing it as a gateway, you never have to worry about HTML versions of *man* pages growing stale as you upgrade your operating system. On the other hand, it

uses CPU cycles to do the HTML conversion each time someone accesses a *man* page through the gateway. The *man2html* kit is available at:

```
ftp://ftp.uci.edu/pub/dtd2html/
```

Look in the directory for the latest *perlWWW tar* file.

Author: Earl Hood <*ehood@convex.com*> at CONVEX Computer Corporation.

ORACLE Gateway

The ORACLE gateway takes a URL query, translates it to an SQL SELECT statement, and returns the results formatted as a table surrounded by <PRE> tags. Although not very polished, this is a a good starting point for providing a hypertext view of an ORACLE database. This gateway requires the ORACLE Pro*C preprocessor. For more information, point your Web browser at:

```
http://info.cern.ch/hypertext/WWW/RDBGate/Implementation.html
```

Author: Arthur Secret.

Genera Gateway

Genera is a gateway used to integrate Sybase databases into the World Wide Web. It can be used to retrofit a Web front end to an existing Sybase database, or to create a new database. *Genera* is more flexible than most gateways because it allows you to develop a custom schema notation that controls the mapping of the Sybase database to the resultant HTML. *Genera* also supports form-based relational querying and whole-database formatting into text and HTML formats. For more information, point your Web browser at:

```
http://cgsc.biology.yale.edu/genera.html
```

Author: Stanley Letovsky <*letovsky@cs.jhu.edu*> at Johns Hopkins Medical School.

20.1.5 Additional cgi-bin Directories

Gateway scripts are usually kept in *cgi-bin*, the CGI binary directory in your ServerRoot directory. If you plan on having dozens of custom gateways developed by different people, you might want to give each person their own *cgi-bin* directory off in their own area of the document tree. If you do this, modify the *srm.conf* configuration file to add the *ScriptAlias* directive for each new *cgi-bin* directory you add to your Web server. For example, to support three cgi-bin directories on your Web server, the Alias directives in the *srm.conf* configuration file might looks like:

```
ScriptAlias /cgi-bin/       /usr/local/etc/httpd/cgi-bin/
ScriptAlias /cs-cgi-bin/    /cs/http/cgi-bin/
ScriptAlias /ee-cgi-bin/    /ee/web/scripts/
```

The ScriptAlias directive is fully described in Appendix G, *Web: access.conf Directives*.

20.2 Forms Processing

Forms are a natural progression from simple queries. If we can collect a search string from the user, why can't we collect a whole form's worth of information?

Forms are officially part of HTML V3.0, but the NCSA Web server provides support for forms. NCSA Mosaic, Lynx, and ViolaWWW support a general-purpose subset of this functionality. Enough functionality is supported to make forms practical and useful.

Let's start with an example. We'll first show you what the final form looks like, then describe the HTML used to generate the form, and finally the CGI script we used to process the results of the form. Figure 20-2 shows a form used to capture user feedback displayed with Mosaic. The users are asked for their name and email address and are given the ability to type an arbitrary amount of feedback text. When done, they can click on Send Comments to terminate the form-entry process. At any point, they can click on Clear Form to reinitialize the form and start over.

When the user terminates the form, the input is sent to the Web server and is passed to a CGI script that formats the input and sends it as a mail message to the Webmaster.

The HTML part of a form is known as the **front end**, while the script that handles the input is known as the **back end**.

20.2.1 HTML for Forms

The form definition is described using HTML and is included as part of the HTML document. The HTML document that displayed the form in Figure 20-2 is:

```
<HTML>
<HEAD>
<TITLE>Crocodile Action Plan Feedback Form</TITLE>
</HEAD>
<BODY>
<IMG SRC="crocodile-thumb.gif">
<H1>Crocodile Action Plan Feedback Form</H1>
<EM>Please send us your comments!</EM>
<P>
<HR>
<P>
<FORM METHOD="POST" ACTION="/cgi-bin/comments">
Name: <INPUT TEXT NAME="feed_name" size=36><BR>
Email: <INPUT TEXT NAME="feed_email" size=36>
<P>
<TEXTAREA NAME="feed_comments" ROWS=8 COLS=40>
</TEXTAREA>
<P>
<INPUT TYPE="submit" VALUE="Send Comments">
<INPUT TYPE="reset" VALUE="Clear Form">
</FORM>
<P>
```

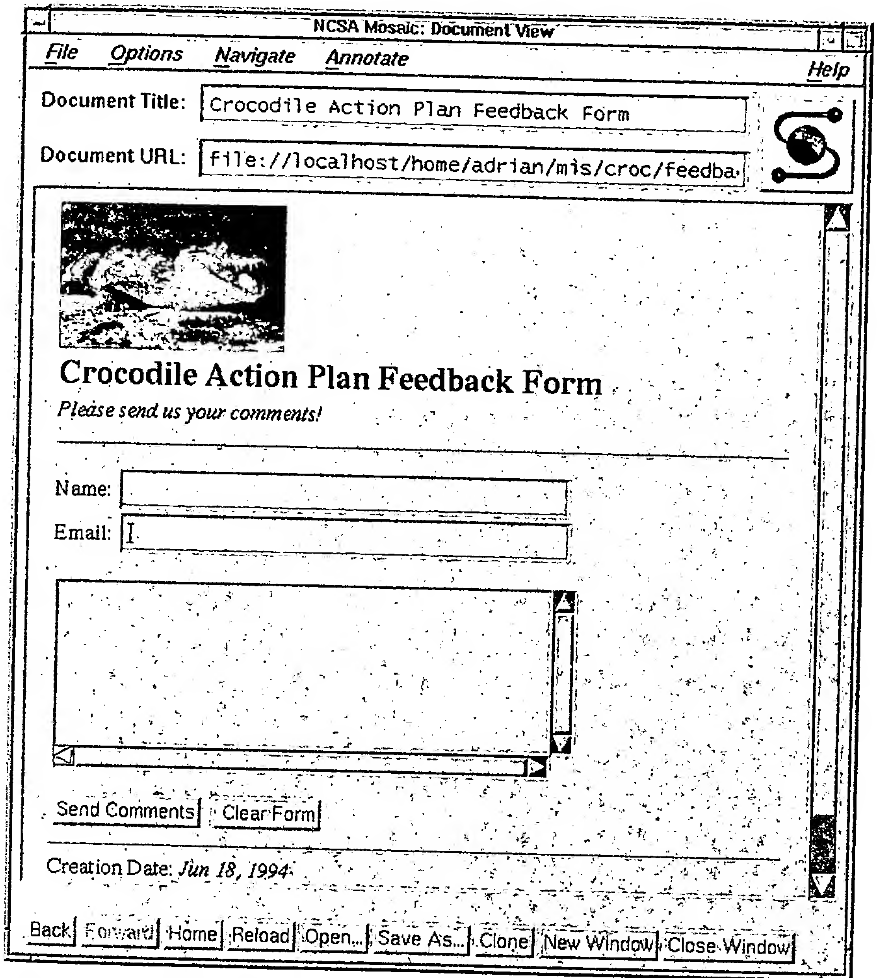


Figure 20-2. Feedback form as displayed by NCSA Mosaic

```
<HR>
<P>
Creation Date: <EM>Jun 18, 1994</EM>
<P>
<ADDRESS><A HREF="rjones.html">DRJ</A></ADDRESS>
</BODY>
</HTML>
```

This HTML is based on the template described in Chapter 19, *Authoring for the Web.* The major new HTML tag introduced in this example is the <FORM> tag. The <FORM> tagging pair is used to enclose the contents of your form and to identify the URL of the CGI script that will process the form input.

AM—Midnight

Inside the <FORM> tagging pair you include other form-related HTML tags and any other HTML construct. You can have multiple forms in a single HTML document. However, you cannot nest forms inside forms.

Constructs that can be in the form include:

- Text entry fields
- Password fields
- Scrollable text areas
- Checkboxes (on/off toggle)
- Radio buttons (one-of-many checkbox toggles)
- Pop-up menus
- Scrollable lists
- Push buttons (submit and reset)

In this example, we provided text entry fields to the user with the <INPUT> tag, and scrollable text areas with the <TEXTAREA> tags. Checkboxes will be demonstrated later in this chapter. For a full list of the form-related HTML tags, see Appendix D, *Web: More HTML Tags.*

20.2.2 Scripts to Handle Form Input

The feedback form uses the POST method to transmit the various fields defined in the form and their associated values back through the CGI to the script */cgi-bin/comments.*

There is no validation at the Web browser level that the input is logically valid or even present. This means that the field-by-field validation process and error handling must happen in your CGI script. This obviously requires a high degree of coordination between the form definition in HTML and the CGI script that processes the form input.*

The next example is the comments CGI script, written in Perl, that we use to process the feedback form. Its function is to take all of the variable input through the CGI, format it, and then send the form contents to the webmaster as an email message.

```
#!/usr/local/bin/perl

$webmaster = "rjones@ix.netcom.com";

#check that POST method is used
if ($ENV{'REQUEST_METHOD'} eq 'POST') {
    # POST method dictates that we get the form
    # input from standard input
```

* However, in-browser validation is on the horizon in HTML 3.0, and some Web browsers, like ViolaWWW, can do it today (although it is difficult to use.)

```
read(STDIN, $buffer, $ENV{'CONTENT_LENGTH'});

# Split the name-value pairs on '&'
@pairs = split(/&/, $buffer);

# Go through the pairs and determine the name
# and value for each form variable.

foreach $pair (@pairs) {
    ($name, $value) = split(/=/, $pair);
    $value =~ tr/+/ /;
    $value =~ s/%([a-fA-F0-9][a-fA-F0-9])/pack("C", hex($1))/eg;
    $FORM{$name} = $value;
}
# Now all the form variables are in the $FORM
# associative array.

# Before proceeding, validate the keyed in email
# for address for evil characters

if ($FORM{EMAIL} !~ /^[a-zA-Z0-9_-+ \t\/@%]+$/) {
    &html_header("Illegal Email Address");
    print "<HR><P>\n";
    print "The Email address you entered contains illegal";
    print "characters. Please back up, correct, then resubmit.\n"
    &html_trailer;
    exit;
}
# The users email address is clean, so open up
# the email message to send the feedback to the
# webmaster and copy the user.
open (MESSAGE,"| mail $webmaster" $FORM{feed_email});

# Format email header information.

print MESSAGE "To: $webmaster\n";
if ($FORM{feed_email} ne "") {
    print MESSAGE "Reply-To: $FORM{feed_email}\n";

}
# Write the actual email message

print MESSAGE "Subject: Comments from $ENV{'REMOTE_HOST'}\n\n";
print MESSAGE "Name: $FORM{feed_name}\n\n";
print MESSAGE "$FORM{feed_comments}\n";
close (MESSAGE);

# Thank the user and acknowledge the feedback

&html_header("Thank You");
print "<HR><P>\n";
print "Your comments have been emailed to the webmaster.\n";
print "You have been copied on the message.\n";
```

AM=Midnight

```
    &html_trailer;
}
else {
    # Format an error message for the user

    &html_header("Comment Form Error");
    print "<HR><P>\n";
    print "Form input was not proccessed.  Please mail your ";
    print "remarks to $webmaster\n";
    &html_trailer;
}

# ====================================================
# This subroutine take a single input parameter and uses
# it as the <TITLE> and the first level header.
# ====================================================
sub html_header {
    $document_title = $_[0];
    print "Content-type: text/html\n\n";
    print "<HTML>\n";
    print "<HEAD>\n";
    print "<TITLE>$document_title</TITLE>\n";
    print "</HEAD>\n";
    print "<BODY>\n";
    print "<H1>$document_title</H1>\n";
    print "<P>\n";
}
# ====================================================
# This subroutine finishes off the HTML stream.
# ====================================================

sub html_trailer{
    print "</BODY>\n";
    print "</HTML>\n";
}
```

The Perl script checks only the email address for possible illegal metacharacters. Although it's not the only variable in the form, it is the only one that is piped directly out to the shell level (in this case with the *mail* command). Note that this script does not accept UUCP-style email addresses, because they include the ! character, which is too dangerous to allow in a Perl script.

The step that formats the mail header does not control to whom the mail message is sent (that is done when the output stream to the *mail* command is opened). However, the mail header is useful so that the Webmaster can reply directly to the email message.

Note especially how the *thank_you* subroutine is used to generate HTML to be sent back and displayed in the user's Web browser. It writes the HTML to *stdout*. When generating HTML from a CGI script, it is absolutely critical that the first line written to *stdout* identify *text/html* MIME-type information. This must be followed by a blank line.

Debugging CGI scripts can be a little tricky. First, make sure your script executes cleanly by itself without syntax errors before you place it in the *cgi-bin* directory and try to execute it by submitting a form. We always check this from the *csh* prior to placing the script or script modification into the *cgi-bin* directory. Otherwise, when we execute the script with the form front end, it may hang and never return control back to the Web browser.

When you run a CGI script from a shell, any run-time errors are reported. You have to emulate the CGI environment by setting up the appropriate environment variables. Otherwise, even if the script is clean, it will fail when it tests whether the REQUEST_METHOD is POST (because that environment variable isn't set) so it will just generate the error HTML back to *stdout*.

20.2.3 Adding Checkboxes to the Form

Our second forms-processing example illustrates how you can use checkboxes with a form. The form registers users who are interested in becoming more involved with crocodile conservation. This example is an extension of both the feedback form and CGI script shown earlier. Figure 20-3 shows what this form looks like when it is displayed using Mosaic.

The HTML used to generate this form started with the HTML from the previous example. Here it is now:

```
<HTML>
<HEAD>
<TITLE>Crocodile Action Plan Registration
Form</TITLE>
</HEAD>
<BODY>
<IMG SRC="crocodile-thumb.gif">
<P>
<EM>I want to be involved in saving endangered crocodiles!</EM>
<P>
<HR>
<P>
<FORM METHOD="POST" ACTION="/cgi-bin/registration">
<INPUT TEXT NAME="reg_name" size=36> Name<BR>
<INPUT TEXT NAME="reg_email" size=36> Email<BR>
<INPUT TEXT NAME="reg_phone" size=36> Phone<BR>
<TEXTAREA NAME="reg_address" ROWS=4 COLS=36></TEXTAREA> Postal
<P>
<DL>
<DT>I would like:</DT>
<DD><INPUT TYPE="checkbox" NAME="reg_catalog"
VALUE="yes">Volunteer work</DD>
<DD><INPUT TYPE="checkbox" NAME="reg_info"
VALUE="yes">
More information on the Species Survival
Commission</DD>
```

```
<P>
<DT>Please contact me about:</DT>
<DD><INPUT TYPE="checkbox" NAME="reg_volunteer"
VALUE="yes">
Volunteer work</DD>
<DD><INPUT TYPE="checkbox" NAME="reg_cash"
VALUE="yes">
Making a monetary contribution to save
endangered crocodiles</DD>
</DL>
<P>
<INPUT TYPE="submit" VALUE="Send Comments">
<INPUT TYPE="reset" VALUE=" Clear Form">
</FORM>
<P>
<HR>
<P>
Creation Date: <EM>Jun 18, 1994</EM>
<P>
<ADDRESS><A HREF="rjones.html">DRJ</A></ADDRESS>
</BODY>
</HTML>
```

To process this form, we copied the */cgi-bin/comments* script used in the first form example to */cgi-bin/registration* and then modified the Perl code that writes the actual email message. The following example shows the modified Perl code:

```
# Write the actual message
print MESSAGE "Subject: Involvement Request \n";
print MESSAGE "Name: $FORM{reg_name}\n\n";

print MESSAGE "Email: $FORM{reg_email}\n\n";
print MESSAGE "Phone: $FORM{reg_phone}\n\n";
print MESSAGE "Address:\n$FORM{reg_address}\n\n";
print MESSAGE "Involvement or Request:\n";
if ($FORM{reg_catalog} eq "yes") {
    print MESSAGE "- I would like a catalog.\n";
}
if ($FORM{reg_info} eq "yes") {
    print MESSAGE "- I would like more information on ";
    print MESSAGE "the Species Survival Commission.\n";
}
if ($FORM{reg_volunteer} eq "yes") {
    print MESSAGE "- I would like to volunteer.\n";
}
if ($FORM{reg_cash} eq "yes") {
    print MESSAGE "- I would like to make a donation.\n";
}
close (MESSAGE);
```

We also modified the "Thank You" message to thank the user appropriately.

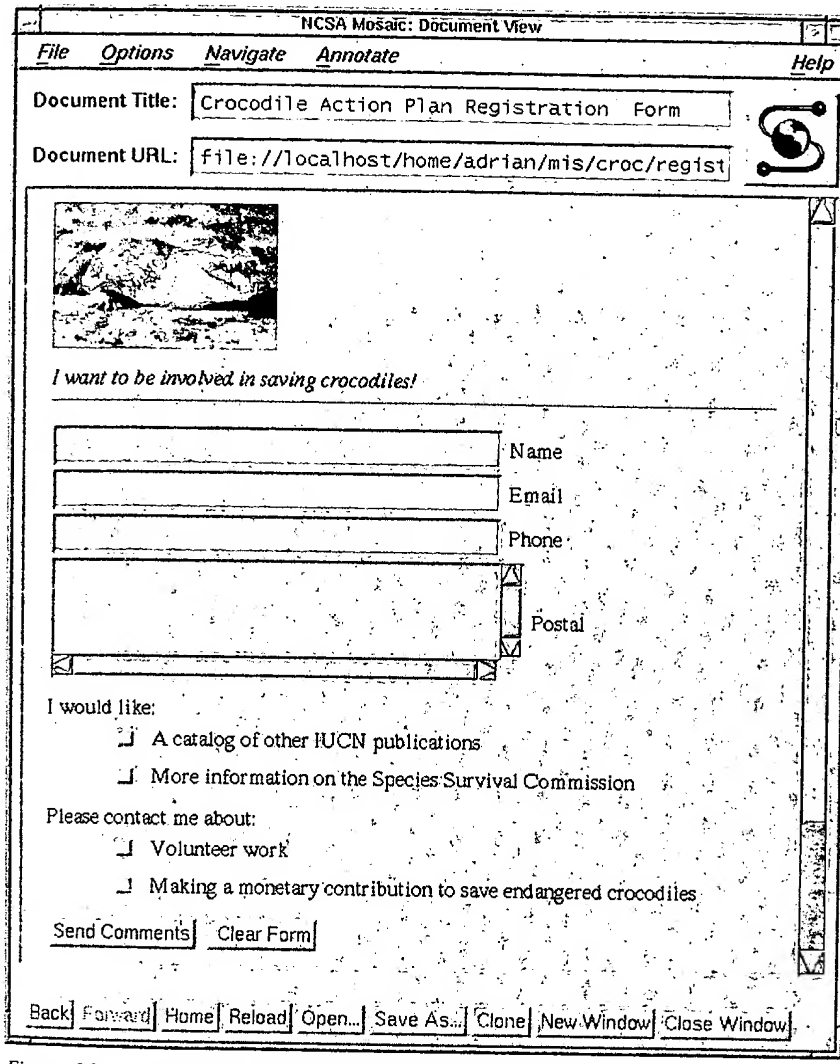


Figure 20-3. Crocodile registration form displayed by NCSA Mosaic

The client/server model used by Web browsers and Web servers is *stateless*. For example, say you wanted to develop a series of three different HTML forms to walk a user through the class registration processing. This seems an obvious use for forms input, but the input captured in the first form is not automatically associated with the input captured in the second form. If a user completes the second form, after having completed the first form, the Web server itself would not know whether the first form had been properly completed or what information was in it. In fact the Web server would have no recollection of handling that first form at all (the server adds a line in a log file, but never looks at it later).

Maintaining state between forms can be done, but your CGI script itself must maintain state information between form transactions. It is not as simple as setting a flag that says "Form 1 has been completed" because you need to keep a record of who completed it. More than one person may be in the process of registering at the same time.

20.3 WAIS Access from the Web

There are four different ways (excuse the pun) that users can access WAIS from the World Wide Web:

- Direct Web browser-to-WAIS-server access. In this method, which doesn't involve your Web server at all, browsers like Mosaic can access WAIS servers directly.
- A public-access Web gateway on the Internet. This is a Web server running on someone else's machine that interacts with WAIS servers and converts the results to HTML.
- A custom WAIS gateway. This is a script or program that is part of your Web server and that interacts with your own WAIS server (or maybe other WAIS servers).
- WAISgate, a commercial Web gateway that provides a forms-based user interface to WAIS queries

All four are viable techniques. Which one you choose depends on whether or not you want to sink time into developing a custom WAIS gateway, how accessible you want your interface to be to the general Web community, or how integrated you want your Web server to be with your WAIS server. The next four sections describe each technique, and Section 20.3.5 summarizes the strengths and weaknesses of each technique.

20.3.1 Direct WAIS Access

Mosaic version 2.0 includes direct WAIS server access. Mosaic operates as a WAIS client, but it displays the results in the HTML window. This approach has nothing to do with your Web server, as all of this prompting and WAIS interaction is done in the Web browser. For example, the following URL directly interacts with a WAIS server from Mosaic:

```
wais://cnidr.org/directory-of-servers
```

When this URL is opened directly, or by hyperlink, the user is prompted for a search query and then the WAIS server at *cnidr.org* processes the query against the *directory-of-servers* WAIS database. Alternatively, the user can specify the search string in the URL using the GET method. For example, in the following URL, the WAIS server running on port 210 at *ds.internic.net* processes the query phrase *MIME* against the *rfcs* WAIS database:

```
wais://ds.internic.net:210/rfcs?MIME
```

To make this direct WAIS access work for searching your data, you must set up a WAIS server and use *waisindex* with the special URL datatype. This directs *waisindex* to build headline files that are formatted as legal URLs. Chapter 8, *Creating WAIS Sources with waisindex*, describes how to use *waisindex*.

As helpful as this direct access is to Mosaic users, using this approach limits your readership to just the Mosaic subset of the Web community. This technique also forces your descriptive headline to be the document URL. This could be confusing to users accessing your WAIS index from dedicated WAIS clients.

20.3.2 Web-to-WAIS Gateways

The oldest technique used to access WAIS servers is to use one of a number of public Web-to-WAIS gateways that are available on the Internet. The two publicly accessible gateways can be accessed with the following URLs:

```
http://info.cern.ch:8001/wais-server-name:port-number/database-name?
http://www.ncsa.uiuc.edu:8001/wais-server-name:port-number/database-name?
```

In practice, both of these appear to be somewhat flaky.

These gateways are actually Web servers that also function as WAIS clients. The Web-to-WAIS gateway interacts with the Web browser to determine the query phrase. Then the gateways execute the query on behalf of the user and take the results of the WAIS search and convert it into HTML on the fly. This is a good way to jump start WAIS/Web interaction. A Web-to-WAIS gateway can be used to access any WAIS database. This technique uses the GET access method described in Section 20.1.1.

20.3.3 Developing a Custom WAIS Gateway

With the advent of the Common Gateway Interface (CGI), it has become much easier to develop your own WAIS gateway. In addition to being able to control the searching function yourself (and generate the resulting HTML yourself), this approach is also desirable because it can be used with any Web browser.

Let's use the same Perl script we used for the *alias-search* example as a starting point. We'll leave the bulk of the structure the same. The user prompting and CGI interaction is the same. The only thing we have to change is the actual search engine, which we will change from an inline regular-expression search to *waisq*. This is the code fragment we will change:

```
# Open the datafile and search through it.
# Write out any matches to stdout.

open(DATAFILE,"$data_file");
while (<DATAFILE>) {
        if (/$query/i) {print;}
}
close(DATAFILE);
```

waisq is the command line interface to WAIS. To invoke the search with the validated search phrase in *$query* we replace the above with:

```
$wais_source = "database-name.src";
$wais_path = "/wais/wais-sources/";
$waisq = "/usr/local/waisq";
$hit_limit = "40";

open(HITS, "-|") || exec ($waisq,"-f","-","-s","$wais_path",
            "-S","$wais_source","-m",$hit_limit,"-g",
            $query);
```

This tells *waisq* that:

–f There is no pre-built question file, so use the source identified by the –S switch and the query identified by the –g switch.

–s $wais_path
The directory where the *.src* file is located

–S $wais_source
The name of the *.src* file

–m $hit_limit
The maximum number of document hits to return

–g $query
The search phrase is in the $query variable

You can play around with these options to use a pre-built WAIS question file. This technique is used to search across multiple WAIS databases with a single query. Using that approach, search results are relevance-ranked within the scope of their respective databases. A search across three different WAIS databases could potentially have three documents all ranked as 1000 (the highest ranking).

In this particular example, after executing the code fragment above, the results of the search are open as an input stream that we can read from HITS. We then loop through that input stream and parse it looking for *:headline*, *:rank*, and *:number-of-lines*.

We then use the associated data to build our own HTML output stream that gets sen back through the Web server to the Web browser that initiated the search. We coul leave the search hits enclosed in the preformatted <PRE> HTML tags, or we could us another type of tagging. We could structure it as a list of bullets, where each entry i the document name. To see an example of how this is done, look at the *wais.pl* cod in the *cgi-bin* directory.

20.3.4 WAISgate

WAISgate is a commercial gateway between the World Wide Web and WAIS. WAISgat allows WAIS database providers to customize entry points into their WAIS databas input forms, search results, and generate HTML documents on the fly. WAISgat

requires a Web browser that handles forms, so it can't be used with all Web browsers. Forms support is required so that WAISgate can generate structured boolean queries. Other features include:

- Access to local and remote WAIS databases
- Single-step processing: search of directory-of-servers, search of a database, and document retrieval
- Multitype documents with associated icons for each document format
- Search by fields in commercial WAIS databases
- Table-lookup conversion from WAIS types to MIME types

WAISgate is available from WAIS, Inc. You can get an online version for general use from:

```
http://www.wais.com/directory-of-servers.html
```

You can't copy or customize this resource, but you can see how WAISgate can be used to reach and search the WAIS directory-of-servers.

20.3.5 Comparison of WAIS-to-Web Techniques

Direct WAIS Access:

- Strength—No development required. Super-clean interface to a local WAIS database.
- Weakness—Only available in Mosaic and then only with direct WAIS patches. Not a general-purpose solution for an extended user community. No customization of search results.

Web-to-WAIS Gateway:

- Strength—No development required. Some degree of integration with your WAIS database and Web server. Public-access Web-to-WAIS gateways already on the Web.
- Weakness—Extensive overhead. WAIS database is often Web-specific and doesn't work with general-purpose WAIS clients. Users typically run into firewall issues. No customization of search results.

Custom WAIS Gateway:

- Strength—You totally control the gateway. Close integration between your WAIS database and Web server. You can customize the search results.
- Weakness—Custom development. When you complain about the gateway, you have to fix it.

WAISgate:

- Strength—Best functional interface to WAIS databases. No development required. Public-access WAISgate interface already on the Web. No firewall issues.
- Weakness—No integration between your WAIS database and your Web server. No customization of search results.

In this Chapter:
- *Access Control and User Authentication*
- *Security*

21

Web: Access Control and Security

You may be happy to give the world unrestricted access to the documentation on your server. If so, the setup described in Chapter 18, *Setting Up a Web Server*, is fine and there is no need for you to read Section 21.1. But you should definitely read Section 21.2, which describes issues involving the security of the system on which your Web server is running.

21.1 Access Control and User Authentication

Access can be controlled using two independent methods:

- Domain-level access control, where rejection or acceptance of connections is based on the Internet address of the system running the Web browser
- User authentication, where rejection or acceptance of connections is based on username and password access authorization. Only a few browsers, like Mosaic, support user authentication.

Access can also be controlled using a combination of these two methods.

21.1.1 Access Control Files

Access control and user authentication can be set up on a server-wide basis or on a directory-by-directory basis:

- Server-wide access and per-directory access is controlled by a global ACF (Access Control File). This ACF is called *access.conf*. The NCSA *httpd* requires that you always set up and administer a global ACF.
- If you choose, per-directory access can also be controlled by per-directory ACFs called *.htaccess* by default.* Per-directory ACFs can be restricted or completely forbidden by the global ACF. You should only use per-directory ACFs if there is a

* The per-directory ACF filename can be changed using the `AccessFileName` directive in the *srm.conf* configuration file.

good reason to decentralize access control (e.g., your server handles multiple independent projects, and each project team needs to handle its own access control).

Per-file access control is not available. If you need to protect an individual file, you should put the file into a directory by itself.

Let's look at how you could control access to two subdirectories in your document tree. With the first approach, subdirectory-1 and subdirectory-2 each has its own <Directory> directive in the global ACF:

```
<Directory /usr/local/etc/httpd/htdocs/subdirectory-1>
        <Limit GET>
        order, deny, allow, & require directives
        </Limit>
</Directory>

<Directory /usr/local/etc/httpd/htdocs/subdirectory-2>
        <Limit GET>
        order, deny, allow, & require directives
        </Limit>
</Directory>
```

If access to each directory needs to be controlled by a different person, both *.htaccess* in subdirectory-1 and *.htaccess* in subdirectory-2 can contain:

```
<Limit GET>
  order, deny, allow, & require directives
</Limit>
```

Outside of the <Limit> section, but within the <Directory> section, in the global ACF, the `Options` directive controls which advanced features you allow on your Web server, and `AllowOverride` defines whether you allow per-directory ACFs to override the global ACF. For more information on these *srm.conf* directives, see Chapter 20, *Web: Gateways and Forms*.

21.1.2 Domain-Level Access Control

You can allow or deny users access to your Web server based on their Internet domain address. This mechanism allows you to control access across an organization or department without regard to specific user names. Domain-level access control is done totally by your Web server, so it works with all Web browsers, unlike authentication.

First, we'll explain a bit about the various directives that control access and then give some examples of various access configurations.

21.1.2.1 Domain control directives

Access control files are sectioned by directory with the <Directory> directive. Within a directory section, the access-control directives are placed within a <Limit> section.

The directives that control access are:

`order` Defines the order in which `deny` and `allow` directives are evaluated within a Limit section

`allow` Defines which hosts can access the directory

`deny` Defines which hosts are denied access to the directory

Let's look at examples of how you alter access control files to satisfy different scenarios.

21.1.2.2 General unrestricted access

The initial Web server configuration described in Chapter 18, *Setting Up a Web Server*, brought your *httpd* server up for general unrestricted access. All users were allowed access to all information. Whether or not this unrestricted access was truly public depends on whether your Web server is running exposed on the Internet, is running behind a firewall, or is on a private TCP/IP network. But your Web server does nothing to restrict access.

21.1.2.3 Local-domain access only

You may be on the Internet or on a private TCP/IP network, but want to restrict access to your local campus or local organization. Assuming you are using the default `DocumentRoot`, just modify your *access.conf* configuration file to look like the following and reboot *httpd*.

```
<Directory /usr/local/etc/httpd/htdocs>
      Options Indexes FollowSymlinks
      AllowOverride None
      <Limit GET>
              order deny,allow
              deny from all
              allow from fredonia.edu
      </Limit>
</Directory>
```

If you compare this against the general unrestricted access configuration you will see that:

- The `order` directive was changed from `allow,deny` to `deny,allow`. This tells `httpd` to evaluate the `deny` directive first and then grant exceptions based on the *allow* directive.
- The `deny from all` directive was added.

- The `allow` directive was changed from `all` to just the local domain (in this case *fredonia.edu*).

You could use the same approach to further restrict access to a subdomain, such as *cs.fredonia.edu*.

You could also extend this approach down to the directory level by including the <Limit> section shown above in an *.htaccess* file in a subdirectory:

```
<Limit GET>
     order deny,allow
     deny from all
     allow from fredonia.edu
</Limit>
```

By putting this directive sequence in an *.htaccess* file in a specific subdirectory, you can have general unrestricted access to your overall Web server, but limit one area to local-domain access only. This technique also works to protect your personal HTML directory, if you place the *.htaccess* file in your personal *public_html* subdirectory.

21.1.2.4 Multi-organization access only

In this scenario, we are creating a private Web on the Internet. This might be for three organizations that are collaborating on a community Web server and want to fully develop the server before opening it up to public access.

If the three organizations are *fredonia.edu*, *non.profit.org*, and *acme.net*, each modifies its respective *access.conf* configuration files to have the same general access control as the previous example, except each would include the following `allow` directive:

```
allow fredonia.edu non.profit.org acme.net
```

This allows anyone from these three domain names to look at the Web server, but denies access to everyone else.

21.1.3 User Authentication

With user authentication, you can allow or deny individuals access to your Web server or document tree directories on a username and password basis. There is no correspondence between your system-level usernames/passwords (in */etc/passwd*) and the Web server's username/password file.

When users access pages that are protected with this mechanism, they are given two prompts (username and password) to which they must respond correctly before access is allowed. Once authenticated, they can navigate from page to page without repeated authentication prompts. This works because the Web browser remembers the hostname, directory path, and name/password for subsequent retrievals. The same name/password pair is reused anytime the Web browser accesses a URL with the same hostname and directory path.

User authentication requires cooperation between a Web browser and your Web server, so unless the Web browser supports user authentication, users cannot be authenticated. Mosaic and Lynx currently support user authentication.

To use user authentication, you have to maintain a private hypertext username/password file. By convention, this file is called *.htpasswd*. User authentication is designed so that a user doesn't need an account on your system in order to be authenticated for access to files on your Web server.

21.1.3.1 Managing the htpasswd file

To manipulate the *.htpasswd* file, you need a program called *htpasswd*. This program is not included with the pre-built binary kit, so you need to get the full source kit and compile it from source. The source is called *htpasswd.c* and is found in the *support/* subdirectory. See Section 18.1.2 for information on how to compile NCSA HTTP (but only follow the directions in the *support/* subdirectory). *htpasswd* is invoked as follows:

```
% htpasswd [-c] .htpasswdusername
```

username is the name of the user you wish to add or edit. The *–c* flag, if present, tells *htpasswd* to create a new hypertext password file instead of editing an existing one.

If *htpasswd* finds the user you specified, it asks you to change the user's password. Type the new password twice. *httpd* then updates the file. If *htpasswd* doesn't find the specified user, it asks you to give the user an initial password.

You can have multiple *.htpasswd* files, or you can use a different filename. The key is that the file must be in a different directory than the one you are protecting and must be located outside the document tree structure.

21.1.3.2 Individual authentication

Individual authentication is accomplished using a combination of access control directives and a private *.htpasswd* file. Let's look at an example of how to restrict access to only individuals knowing the password.

Assuming you are using the default DocumentRoot, just modify your *access.conf* configuration file to look like the following.

```
<Directory /usr/local/etc/httpd/htdocs>
      Options Indexes FollowSymlinks
      AllowOverride None
      AuthUserFile /usr/local/etc/httpd/conf/.htpasswd
      AuthGroupFile /dev/null
      AuthName By Secret Password Only!
      AuthType Basic
      <Limit GET>
            require user username
      </Limit>
</Directory>
```

If you compare this to the general unrestricted access configuration you will see that:

- The `AuthUserFile` directive is added to specify the absolute pathname to the hypertext password file.
- The `AuthGroupFile` directive has been added, but set to */dev/null*, which by UNIX standards indicates that the file doesn't exist.
- The `AuthName` directive is added to specify the prompt to be given to the user for the username (in this case `By Secret Password Only!`).
- The `AuthType` directive is added and set to `Basic`. You currently don't have a choice about this, as `Basic` is the only authorization type available.
- All four `Auth` directives go outside the Limit sectioning directive.
- The `order` and `allow` directives were removed from the <Limit> sectioning directive and replaced with the `require` directive. This tells *httpd* to prompt for a username and password and that the username has to be *username*. Using the `require` directive dictates the need to include the `AuthUserFile`, `AuthGroupFile`, `AuthName`, and `AuthType` directives.

Next, create a hypertext password file for the username specified in your *access.conf* configuration file:

```
% htpasswd -c /usr/local/etc/httpd/conf/.htpasswd username
```

htpasswd prompts you for the password. Remember that for any changes to the configuration files (including *.htpasswd*) to take effect, you must reboot *httpd* as described in Section 18.1.5.2 (unless you are running the server under *inetd*). After your Web server restarts, access is restricted to only those individuals that know the user name and the accompanying password.

If you want to use individual authentication on a directory-level basis to protect the */usr/local/etc/httpd/htdocs/subdirectory-3* directory, you need to place the following directives in a *.htaccess* file in that directory:

```
AuthUserFile /usr/local/etc/httpd/conf/.htpasswd
AuthGroupFile /dev/null
AuthName By Secret Password Only!
AuthType Basic
<Limit GET>
     require user username
</Limit>
```

21.1.3.3 Group authentication

In addition to authenticating users individually, you can also place them into groups and then treat each group as a whole. There are three steps to this process:

- Modify the global ACF (*access.conf*) file.

- Create a hypertext group file (*.htgroup*) with multiple users as members.
- Double-check that all the users in the hypertext group file are also in the hypertext password file.

Let's look at a specific example. Using the previous global ACF we used for individual authentication as a starting point, you would modify it to look like this:

```
<Directory /usr/local/etc/httpd/htdocs>
     Options Indexes FollowSymlinks
     AllowOverride None
     AuthUserFile /usr/local/etc/httpd/conf/.htpasswd
     AuthGroupFile /usr/local/etc/httpd/conf/.htgroup
     AuthName By Secret Password Only!
     AuthType Basic
     <Limit GET>
          require group groupname
     </Limit>
</Directory>
```

If you compare this against the starting point, you will see that:

- The `AuthGroupFile` directive was modified from */dev/null* to point to a group password file called *.htgroup*. It's a good idea to place this file in the same directory as the hypertext password file.
- The `require` directive was modified from `user` to `group` and the *`username`* was changed to *`groupname`*.

Then, in the second step, you create the */usr/local/etc/httpd/conf/.htgroup* file with a text editor and place the following group definition in it:

```
groupname: username1 username2 username3 usernameN
```

Usernames are separated by spaces. Multiple groups could be identified in the hypertext group file, one per line.

If you have already defined the usernames you associated with the group name, then you just need to reboot *httpd* to have these changes take effect.

If you added usernames to the group file that are not entered into the hypertext password file, then you use the *htpasswd* program to add usernames and associated passwords to the existing *.htpasswd* file. For example,

```
% htpasswd /usr/local/etc/httpd/conf/.htpasswd username2
% htpasswd /usr/local/etc/httpd/conf/.htpasswd username3
```

Note that *htpasswd* is invoked without the *–c* flag, since the hypertext password file already exists and these new usernames are just being added to it.

21.1.3.4 Simultaneous access control and user authentication

User authentication can be used in conjunction with domain-level access control. For example, in either of the two preceding examples, you could have specified that only users from the *fredonia.edu* domain could get to the user authentication prompts. To do this, you would modify the <Limit> section to look like:

```
<Limit GET>
      order deny,allow
      deny from all
      allow from fredonia.edu
      require group groupname
</Limit>
```

Using the two together adds another level of security to your Web server.

21.2 Security

When configuring access control for your Web server, you want to make sure you do not give unauthorized access to anyone who may deliberately try to harm your system. Your Web server is secure if you set it up as described in Chapter 18, *Setting Up a Web Server*, for unrestricted access. However, as you incorporate advanced features into your Web server such as gateways, you should also incorporate a heightened sense of caution.

All of the following security concerns should be tempered by how your system is connected to the Internet. If your system is protected behind a firewall, your security concerns are different than if you are directly attached to the Internet.

21.2.1 Links Outside the Document Tree

FTP and Gopher both have the `chroot()` feature, wherein access is absolutely restricted to files within the data tree. Web servers don't have this feature. However, Web browsers can't randomly walk across the directory structure on the system. All access is relative to the top of the document tree (as defined by the DocumentRoot directive). Even when looking through a directory with automatically generated indexes, *httpd* does not let users change directories up past the top of the document tree.

However, unlike FTP and Gopher (when set up properly), links from within the document tree to outside the tree do work. The biggest danger for a webmaster is a system with users who might create a symbolic link to a directory outside the document tree. This could happen either in the document tree itself or in a user's private HTML directory. To safeguard against this, the webmaster should either disable the *followsymlink* option in *access.conf* or at least change it to *SymLinksIfOwnerMatch*.

21.2.2 Access Control and User Authentication

Access control and user authentication make your Web server relatively safe, but not bulletproof:

- With domain-based access control, your Web server is only as safe as DNS, the Domain Name Server. If you trust DNS, you should trust domain-based access control.
- With user authentication, the password is passed over the network in *uuencoded* format (simple ASCII encoding not to be confused with encryption). This method is similar to Telnet-style username and password security. So if you trust your machine to be on the Internet (open to Telnet attempts by anyone), then you have no reason not to trust this method as well.

The bottom line is that if you absolutely cannot have information be seen by outside people, you probably should not use the NCSA Web server (or any other current Web server).

21.2.3 Personal HTML Directories

When configuring your Web server, if you do nothing with the UserDir directive in the *srm.conf* configuration file, your Web server provides public access to a *public_html* directory under each user's home directory. Some users on your system will not even be aware of this, in which case there is no problem. Others will want to serve HTML documents from this directory. These are the users that need protection and, sometimes, the ones you need to protect against.

If all of your users have their home directories in one physical location (such as */home*), protection is easy. You include the following directives in your *access.conf* configuration file:

```
<Directory /home>
    AllowOverride None
    Options Indexes
</Directory>
```

If they are not all in one location, then you should use a wildcard pattern matching the name of the UserDir directive to protect the directories:

```
<Directory /*/public_html*>
    AllowOverride None
    Options Indexes
</Directory>
```

Setting the AllowOverride directive to None and the Options directive to Indexes prevents users from accidentally executing CGI scripts with potential security problems, purposely including executable CGI scripts in the HTML stream, or establishing symbolic links that lead out of the document tree.

If you want to give your users the ability to create symbolic links, at least add the `SymLinksIfOwnerMatch` clause to the `Options` directive to assure that they can only establish symbolic links to files and directories they own.

21.2.4 Shell Metacharacters in Forms

The addition of interactive forms gives users the ability to key in shell metacharacters in form input fields in an attempt to create shell syntax errors or execute arbitrary commands on your system. These must be trapped in the CGI scripts associated with each form. This problem and how to protect against it are described in detail in Section 20.1.2.2.

CGI scripts can be placed in three locations, the main *cgi-bin* subdirectory, alternative *cgi-bin* subdirectories (that you establish with the `ScriptAlias directive`), and in users' personal HTML subdirectories. You will have to either police all of these areas or else have confidence in your user community to not expose security holes in their personal CGI scripts. If you can't trust them to do this, you simply make sure that the Options directive used to protect users' personal HTML directories is not set to `All` or `ExecCGI`. The best way to handle users who can't be trusted is to specify the Options directive in *access.conf* to be used server-wide and then use `AllowOverride None`.

21.2.5 Server-Side Includes

Server-side includes have not been discussed in detail, but can be a potentially dangerous security issue. Similar to an untrusted CGI script, server-side includes can be used to write anything into the HTML stream going back to a Web browser. It is generally a good idea to completely disable server-side includes wherever possible. Not only will you be plugging a potential security hole, but you will also be getting a big performance gain on your Web server.

Server-side includes are enabled, server-wide or per-directory, by setting the `Options` directive to either `All` or `Includes`. If you do allow includes, particularly in user's personal HTML directories, at least make sure you set the `Options` directive to `IncludesNoExec`, which does not allow executable CGI scripts to be included for execution in the HTML stream.